salabim Documentation

Release 21.1.7

Ruud van der Ham

Oct 13, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

Salabim is a package for discrete event simulation in Python. It follows the methodology of process description as originally demonstrated in *Simula* and later in *Prosim*, *Must* and *Tomas*. The process interaction methods are also quite similar to *SimPy 2*.

The package comprises discrete event simulation, queue handling, resources, statistical sampling and monitoring. On top of that real time animation is built in.

The package comes with a number of sample models.

1.1 Requirements

Salabim runs on

- CPython
- PyPy platform
- Pythonista (iOS)
- PyDroid3 (Android)

The package is runs only under Python 3.6 and higher.

The following packages are required:

Platform	Animation	3D animation	Video (mp4, avi)	Animated GIF/PNG
CPython	Pillow	Pillow, pyopengl	opencv, numpy	Pillow
PyPy	Pillow	?	opencv, numpy	Pillow
Pythonista	Pillow	N/A	N/A	Pillow
PyDroid3	Pillow	N/A	opencv, numpy	Pillow

Several CPython packages, like *WinPython* support Pillow out of the box. If not, install with

```
pip install Pillow
```

Under Linux, Pillow can be installed with

```
sudo apt-get purge python3-pil
sudo apt-get install python3-pil python3-pil.imagetk
```

For 3D animation, installation of OpenGL is required

```
pip install pyopengl
pip install pyopengl_accelerate
```

If, after that you get a runtime error that glutInit is not defined, try to install from the *Unofficial Windows Binaries for Python Extension Packages* site at https://www.lfd.uci.edu/~gohlke/pythonlibs/#pyopengl

Then, find the right 3.1.5 or later version for you (e.g. for Python 3.7, 64 bits you use `PyOpenGL-3.1.5-cp37-cp37m-win_amd64.whl`) and download that. Then issue

```
pip install wheelfile
```

,like (for the above package)

```
pip install PyOpenGL-3.1.5-cp37-cp37m-win_amd64.whl
```

and it should work.

It is recommended to install pyopengl_accelerate in the same way, like

```
pip install PyOpenGL_accelerate-3.1.5-cp37-cp37m-win_amd64.whl
```

Under Linux (at least Debian), use

```
pip3 install pyopengl
sudo apt-get install freeglut3-dev
```

to install PyOpenGL.

In order to use .obj files with 3D animations, pywavefront and pyglet are required

```
pip install pywavefront
pip install pyglet``
```

For video production, installation of opencv and numpy may be required with

```
pip install opencv-python
pip install numpy
```

To add audio to a video (Windows only), installation of ffmpeg is required. Refer to *www.ffmpeg.org* for instructions and download.

Running models under PyPy is highly recommended for production runs, where run time is important. We have found 6 to 7 times faster execution compared to CPython. However, for development, nothing can beat CPython or Pythonista.

1.2 Installation

The preferred way to install salabim is from PyPI with

```
pip install salabim
```

or to upgrade to a new version

```
pip install salabim --upgrade
```

Alternatively, it is possible to install salabim directly from GitHub with the utility *install salabim from github.py*, which can be found in the GitHub repository.

You can find the package along with some support files and sample models on www.github.com/salabim/salabim. From there you can directly download as a zip file and next extract all files. Alternatively the repository can be cloned.

For Pythonista, the easiest way to download salabim is:

- Tap 'Open in…'.
- Tap 'Run Pythonista Script'.
- Pick this script and tap the run button
- Import file
- Possibly after short delay, there will be a salabim-master.zip file in the root directory
- Tap this zip file and Extract files

- All files are now in a directory called salabim-master
- Optionally rename this directory to salabim

Salabim itself is provided as one Python script, called salabim.py. You may place that file in any directory where your models reside.

If you want salabim to be available from other directories, without copying the salabim.py script, either install from PyPI (see above) or run the supplied *install salabim.py* file or *install salabim from github.py*. In doing so, you will create (or update) a salabim directory in the site-package directory, which will then contain a copy of the salabim package.

1.3 Python

Python is a widely used high-level programming object oriented language for general-purpose programming, created by Guido van Rossum and first released in 1991. An interpreted language, Python has a design philosophy that emphasizes code readability (notably using whitespace indentation to delimit code blocks rather than curly brackets or keywords), and a syntax that allows programmers to express concepts in fewer lines of code than might be used in languages such as C++ or Java. The language provides constructs intended to enable writing clear programs on both a small and large scale.

A good way to start learning about Python is https://www.python.org/about/gettingstarted/

CHAPTER TWO

MODELING

2.1 A simple model

Let's start with a very simple model, to demonstrate the basic structure, process interaction, component definition and output:

```python
# Car.py
import salabim as sim


class Car(sim.Component):
    def process(self):
        while True:
            yield self.hold(1)


env = sim.Environment(trace=True)
Car()
env.run(till=5)
```

In basic steps:

We always start by importing salabim

```python
import salabim as sim
```

Now we can refer to all salabim classes and function with `sim.`. For convenience, some functions or classes can be imported with, for instance

```
from salabim import now, main, Component
```

It is also possible to import all methods, classes and globals by

```
from salabim import *
```

, but we do not recommend that method.

The main body of every salabim model usually starts with

```
env = sim.Environment()
```

For each (active) component we define a class as in

```
class Car(sim.Component):
```

The class inherits from sim.Component.

Although it is possible to define other processes within a class, the standard way is to define a generator function called `process` in the class. A generator is a function with at least one yield statement. These are used in salabim context as a signal to give control to the sequence mechanism.

In this example,

```
yield self.hold(1)
```

gives control,to the sequence mechanism and *comes back* after 1 time unit. The *self.* part means that it is this component to be held for some time. We will see later other uses of yield like passivate, request, wait and standby.

In the main body an instance of a car is created by Car(). It automatically gets the name car.0. As there is a generator function called process in Car, this process description will be activated (by default at time now, which is 0 here). It is possible to start a process later, but this is by far the most common way to start a process.

With

```
env.run(till=5)
```

we start the simulation and get back control after 5 time units. A component called *main* is defined under the hood to get access to the main process.

When we run this program, we get the following output

```
line#         time current component    action                               information
-----   ---------- -------------------- -----------------------------------  ------------------------------------------------
                                        line numbers refers to               Example - basic.py
```

```
  11                              default environment initialize
  11                              main create
  11         0.000 main           current
  12                              car.0 create
  12                              car.0 activate                  scheduled for      0.000 @    6  process=process
  13                              main run                        scheduled for      5.000 @   13+
   6         0.000 car.0          current
   8                              car.0 hold                      scheduled for      1.000 @    8+
   8+        1.000 car.0          current
   8                              car.0 hold                      scheduled for      2.000 @    8+
   8+        2.000 car.0          current
   8                              car.0 hold                      scheduled for      3.000 @    8+
   8+        3.000 car.0          current
   8                              car.0 hold                      scheduled for      4.000 @    8+
   8+        4.000 car.0          current
   8                              car.0 hold                      scheduled for      5.000 @    8+
  13+        5.000 main           current
```

2.2 A bank example

Now let's move to a more realistic model. Here customers are arriving in a bank, where there is one clerk. This clerk handles the customers in first in first out (FIFO) order. We see the following processes:

- The customer generator that creates the customers, with an inter arrival time of uniform(5,15)
- The customers
- The clerk, which serves the customers in a constant time of 30 (overloaded and non steady state system)

And we need a queue for the customers to wait for service.

The model code is:

```python
# Bank, 1 clerk.py
import salabim as sim


class CustomerGenerator(sim.Component):
    def process(self):
        print(env.main().scheduled_time())
        while True:
            Customer()
            yield self.hold(sim.Uniform(5, 15).sample())


class Customer(sim.Component):
    def process(self):
        self.enter(waitingline)
        if clerk.ispassive():
            clerk.activate()
        yield self.passivate()


class Clerk(sim.Component):
    def process(self):
        while True:
            while len(waitingline) == 0:
                yield self.passivate()
            self.customer = waitingline.pop()
            yield self.hold(30)
            self.customer.activate()


env = sim.Environment(trace=True)

CustomerGenerator()
clerk = Clerk()
waitingline = sim.Queue("waitingline")

env.run(till=50)
print()
waitingline.print_statistics()
```

Let's look at some details

```
yield self.hold(sim.Uniform(5, 15).sample())
```

will do the statistical sampling and wait for that time till the next customer is created.

With

```
self.enter(waitingline)
```

the customer places itself at the tail of the waiting line.

Then, the customer checks whether the clerk is idle, and if so, activates him immediately.

```
if clerk.ispassive():
    clerk.activate()
```

Once the clerk is active (again), it gets the first customer out of the waitingline with

```
self.customer = waitingline.pop()
```

and holds for 30 time units with

```
yield self.hold(30)
```

After that hold the customer is activated and will terminate

```
self.customer.activate()
```

In the main section of the program, we create the CustomerGenerator, the Clerk and a queue called waitingline. After the simulation is finished, the statistics of the queue are presented with

```
waitingline.print_statistics()
```

The output looks like

```
line#         time current component    action                               information
-----   ---------- -------------------- -----------------------------------  ------------------------------------------------
                                        line numbers refers to               Example - bank, 1 clerk.py
   30                                   default environment initialize
   30                                   main create
   30        0.000 main                 current
   32                                   customergenerator create
   32                                   customergenerator activate           scheduled for      0.000 @    6  process=process
   33                                   clerk.0 create
   33                                   clerk.0 activate                     scheduled for      0.000 @   21  process=process
   34                                   waitingline create
   36                                   main run                             scheduled for     50.000 @   36+
    6        0.000 customergenerator    current
    8                                   customer.0 create
    8                                   customer.0 activate                  scheduled for      0.000 @   13  process=process
    9                                   customergenerator hold               scheduled for     14.631 @    9+
   21        0.000 clerk.0              current
   24                                   clerk.0 passivate
   13        0.000 customer.0           current
   14                                   customer.0                           enter waitingline
   16                                   clerk.0 activate                     scheduled for      0.000 @   24+
   17                                   customer.0 passivate
   24+       0.000 clerk.0              current
   25                                   customer.0                           leave waitingline
   26                                   clerk.0 hold                         scheduled for     30.000 @   26+
    9+      14.631 customergenerator    current
    8                                   customer.1 create
    8                                   customer.1 activate                  scheduled for     14.631 @   13  process=process
    9                                   customergenerator hold               scheduled for     21.989 @    9+
   13       14.631 customer.1           current
   14                                   customer.1                           enter waitingline
   17                                   customer.1 passivate
    9+      21.989 customergenerator    current
    8                                   customer.2 create
    8                                   customer.2 activate                  scheduled for     21.989 @   13  process=process
    9                                   customergenerator hold               scheduled for     32.804 @    9+
   13       21.989 customer.2           current
   14                                   customer.2                           enter waitingline
   17                                   customer.2 passivate
   26+      30.000 clerk.0              current
   27                                   customer.0 activate                  scheduled for     30.000 @   17+
   25                                   customer.1                           leave waitingline
   26                                   clerk.0 hold                         scheduled for     60.000 @   26+
   17+      30.000 customer.0           current
                                        customer.0 ended
    9+      32.804 customergenerator    current
    8                                   customer.3 create
    8                                   customer.3 activate                  scheduled for     32.804 @   13  process=process
    9                                   customergenerator hold               scheduled for     40.071 @    9+
   13       32.804 customer.3           current
   14                                   customer.3                           enter waitingline
   17                                   customer.3 passivate
```

```
    9+      40.071 customergenerator    current
    8                                   customer.4 create
    8                                   customer.4 activate                  scheduled for     40.071 @   13  process=process
    9                                   customergenerator hold               scheduled for     54.737 @    9+
   13       40.071 customer.4           current
   14                                   customer.4                           enter waitingline
   17                                   customer.4 passivate
   36+      50.000 main                 current

Statistics of waitingline at        50
                                                                    all    excl.zero         zero
-------------------------------------------- -------------- ------------- ------------- -------------
Length of waitingline                        duration            50          35.369       14.631
                                             mean                 1.410        1.993
                                             std.deviation        1.107        0.754

                                             minimum              0            1
                                             median               2            2
                                             90% percentile       3            3
                                             95% percentile       3            3
                                             maximum              3            3

Length of stay in waitingline                entries              2            1            1
                                             mean                 7.684       15.369
                                             std.deviation        7.684        0

                                             minimum              0           15.369
                                             median              15.369       15.369
                                             90% percentile      15.369       15.369
                                             95% percentile      15.369       15.369
                                             maximum             15.369       15.369
```

Now, let's add more clerks. Here we have chosen to put the three clerks in a list

```
clerks = [Clerk() for _ in range(3)]
```

although in this case we could have also put them in a salabim queue, like

```
clerks = sim.Queue('clerks')
for _ in range(3):
    Clerk().enter(clerks)
```

And, to restart a clerk

```
for clerk in clerks:
    if clerk.ispassive():
       clerk.activate()
       break  # reactivate only one clerk
```

The complete source of a three clerk post office:

```
# Bank, 3 clerks.py
import salabim as sim


class CustomerGenerator(sim.Component):
    def process(self):
        while True:
            Customer()
            yield self.hold(sim.Uniform(5, 15).sample())


class Customer(sim.Component):
    def process(self):
        self.enter(waitingline)
        for clerk in clerks:
            if clerk.ispassive():
                clerk.activate()
                break  # activate at most one clerk
        yield self.passivate()


class Clerk(sim.Component):
    def process(self):
        while True:
            while len(waitingline) == 0:
                yield self.passivate()
            self.customer = waitingline.pop()
            yield self.hold(30)
            self.customer.activate()
```

```
env = sim.Environment(trace=False)
CustomerGenerator()
clerks = [Clerk() for _ in range(3)]

waitingline = sim.Queue("waitingline")

env.run(till=50000)
waitingline.print_histograms()

waitingline.print_info()
```

2.3 A bank office example with resources

The salabim package contains another useful concept for modelling: resources. Resources have a limited capacity and can be claimed by components and released later.

In the model of the bank with the same functionality as the above example, the clerks are defined as a resource with capacity 3.

The model code is:

```
# Bank, 3 clerks (resources).py
import salabim as sim


class CustomerGenerator(sim.Component):
    def process(self):
        while True:
            Customer()
            yield self.hold(sim.Uniform(5, 15).sample())


class Customer(sim.Component):
    def process(self):
        yield self.request(clerks)
        yield self.hold(30)
        self.release()  # not really required
```

```
env = sim.Environment(trace=False)
CustomerGenerator()
clerks = sim.Resource("clerks", capacity=3)

env.run(till=50000)

clerks.print_statistics()
clerks.print_info()
```

Let's look at some details.

```
clerks = sim.Resource('clerks', capacity=3)
```

This defines a resource with a capacity of 3.

And then, a customer, just tries to claim one unit (=clerk) from the resource with

```
yield self.request(clerks)
```

Here, we use the default of 1 unit. If the resource is not available, the customer just waits for it to become available (in order of arrival).

In contrast with the previous example, the customer now holds itself for 30 time units.

And after these 30 time units, the customer releases the resource with

```
self.release()
```

The effect is that salabim then tries to honor the next pending request, if any.

(actually, in this case this release statement is not required, as resources that were claimed are automatically released when a process terminates).

The statistics are maintained in two system queue, called clerk.requesters() and clerk.claimers().

The output is very similar to the earlier example. The statistics are exactly the same.

2.4 The bank office example with balking and reneging

Now, we assume that clients are not going to the queue when there are more than 5 clients waiting (balking). On top of that, if a client is waiting longer than 50, he/she will leave as well (reneging).

The model code is:

```
# Example - bank, 3 clerks, reneging.py
import salabim as sim


class CustomerGenerator(sim.Component):
    def process(self):
        while True:
            Customer()
            yield self.hold(sim.Uniform(5, 15).sample())


class Customer(sim.Component):
    def process(self):
        if len(waitingline) >= 5:
            env.number_balked += 1
            env.print_trace("", "", "balked")
            yield self.cancel()
        self.enter(waitingline)
        for clerk in clerks:
            if clerk.ispassive():
                clerk.activate()
                break  # activate only one clerk
        yield self.hold(50)  # if not serviced within this time, renege
        if self in waitingline:
            self.leave(waitingline)
            env.number_reneged += 1
            env.print_trace("", "", "reneged")
        else:
            yield self.passivate()  # wait for service to be completed


class Clerk(sim.Component):
    def process(self):
        while True:
            while len(waitingline) == 0:
                yield self.passivate()
            self.customer = waitingline.pop()
            self.customer.activate()  # get the customer out of it's hold(50)
            yield self.hold(30)
            self.customer.activate()  # signal the customer that's all's done
```

```
env = sim.Environment()
CustomerGenerator()
env.number_balked = 0
env.number_reneged = 0
clerks = [Clerk() for _ in range(3)]

waitingline = sim.Queue("waitingline")
waitingline.length.monitor(False)
env.run(duration=1500)  # first do a prerun of 1500 time units without collecting data
waitingline.length.monitor(True)
env.run(duration=1500)  # now do the actual data collection for 1500 time units
waitingline.length.print_histogram(30, 0, 1)
print()
waitingline.length_of_stay.print_histogram(30, 0, 10)
print("number reneged", env.number_reneged)
print("number balked", env.number_balked)
```

Let's look at some details.

```
yield self.cancel()
```

This makes the current component (a customer) a data component (and be subject to garbage collection), if the queue length is 5 or more.

The reneging is implemented by a hold of 50. If a clerk can service a customer, it will take the customer out of the waitingline and will activate it at that moment. The customer just has to check whether he/she is still in the waiting line. If so, he/she has not been serviced in time and thus will renege.

```
yield self.hold(50)
if self in waitingline:
    self.leave(waitingline)
    env.number_reneged += 1
else:
     self.passivate()
```

All the clerk has to do when starting servicing a client is to get the next customer in line out of the queue (as before) and activate this customer (at time now). The effect is that the hold of the customer will end.

```
self.customer = waitingline.pop()
self.customer.activate()
```

2.5 The bank office example with balking and reneging (resources)

Now we show how the balking and reneging is implemented with resources.

The model code is:

```
# Example - bank, 3 clerks, reneging (resources).py
import salabim as sim


class CustomerGenerator(sim.Component):
    def process(self):
        while True:
            Customer()
            yield self.hold(sim.Uniform(5, 15).sample())


class Customer(sim.Component):
    def process(self):
        if len(clerks.requesters()) >= 5:
            env.number_balked += 1
            env.print_trace("", "", "balked")
            yield self.cancel()
        yield self.request(clerks, fail_delay=50)
        if self.failed():
            env.number_reneged += 1
            env.print_trace("", "", "reneged")
        else:
            yield self.hold(30)
            self.release()


env = sim.Environment()
CustomerGenerator()
env.number_balked = 0
env.number_reneged = 0
clerks = sim.Resource("clerks", 3)

env.run(till=50000)

clerks.requesters().length.print_histogram(30, 0, 1)
```

```
print()
clerks.requesters().length_of_stay.print_histogram(30, 0, 10)
print("number reneged", env.number_reneged)
print("number balked", env.number_balked)
```

As you can see, the balking part is exactly the same as in the example without resources.

For the renenging, all we have to do is add a fail_delay

```
yield self.request(clerks, fail_delay=50)
```

If the request is not honored within 50 time units, the process continues after that request statement. And then, we just check whether the request has failed

```
if self.failed():
    env.number_reneged += 1
```

This example shows clearly the advantage of the resource solution over the passivate/activate method, in this example.

2.6 The bank office example with states

The salabim package contains yet another useful concept for modelling: states. In this case, we define a state called worktodo.

The model code is:

```
# Example - bank, 3 clerks (state).py
import salabim as sim


class CustomerGenerator(sim.Component):
    def process(self):
        while True:
            Customer()
            yield self.hold(sim.Uniform(5, 15).sample())


class Customer(sim.Component):
    def process(self):
        self.enter(waitingline)
        worktodo.trigger(max=1)
        yield self.passivate()
```

```
class Clerk(sim.Component):
    def process(self):
        while True:
            if len(waitingline) == 0:
                yield self.wait((worktodo, True, 1))
            self.customer = waitingline.pop()
            yield self.hold(30)
            self.customer.activate()


env = sim.Environment()
CustomerGenerator()
for i in range(3):
    Clerk()
waitingline = sim.Queue("waitingline")
worktodo = sim.State("worktodo")

env.run(till=50000)
waitingline.print_histograms()
worktodo.print_histograms()
```

Let's look at some details.

```
worktodo = sim.State('worktodo')
```

This defines a state with an initial value False.

In the code of the customer, the customer tries to trigger one clerk with

```
worktodo.trigger(max=1)
```

The effect is that if there are clerks waiting for worktodo, the first clerk's wait is honored and that clerk continues its process after

```
yield self.wait(worktodo)
```

Note that the clerk is only going to wait for worktodo after completion of a job if there are no customers waiting.

2.7 The bank office example with standby

The salabim package contains yet another powerful process mechanism, called standby. When a component is in standby mode, it will become current after *each* event. Normally, the standby will be used in a while loop where at every event one or more conditions are checked.

The model with standby is

```
.. literalinclude:: ..\..\sample models\Bank, 3 clerks (standby.py)
```

In this case, the condition is checked frequently with

```
while len(waitingline) == 0:
    yield self.standby()
```

The rest of the code is very similar to the version with states.

Warning: It is very important to realize that this mechanism can have significant impact on the performance, as after EACH event, the component becomes current and has to be checked. In general it is recommended to try and use states or a more straightforward passivate/activate construction.

CHAPTER

THREE

COMPONENT

Components are the key elements of salabim simulations.

Components can be either data or active. An active component has one or more process descriptions and is activated at some point of time. You can make a data component active with activate. And an active component can become data either with a cancel or by reaching the end of its process method.

It is easy to create a data component by:

```
data_component = sim.Component()
```

Data components may be placed in a queue. This component will not be activated as there is no associated process method.

In order to make an active component it is necessary to first define a class:

```
class Ship(sim.Component):
```

And then there has to be a (usually generator) method, normally called process:

```
class Ship(sim.Component):
    def process(self):
        ....
        yield ...
        ....
```

Normally, the process will contain at least one yield (or yield from) statement. But that's not a requirement.

Creation and activation can be combined by making a new instance of the class:

```
ship1 = Ship()
ship2 = Ship()
ship3 = Ship()
```

This causes three Ships to be created and to start them at Sim.process(). The ships will automatically get the name `ship.0`, etc., unless a name is given explicitly.

If no process method is found for Ship, the ship will be a data component. In that case, it may become active by means of an activate statement:

```
class Crane(sim.Component):
    def unload(self):
        ....
        yield ...
        ....

crane1 = Crane()
crane1.activate(process='unload')

crane2 = Crane(process='unload')
```

Effectively, creation and start of crane1 and crane2 is the same.

Although not very common, it is possible to activate a component at a certain time or with a specified delay:

```
ship1.activate(at=100)
ship2.activate(delay=50)
```

At time of creation it is sometimes useful to be able to set attributes, prepare for actions, etc. This is possible in salabim by defining an __init__ and/or a setup method:

If the __init__ method is used, it is required to call the Component.__init__ method from within the overridden method:

```
class Ship(sim.Component):
    def __init__(self, length, *args, **kwargs):
        sim.Component.__init__(self, *args, **kwargs)
        self.length = length

ship = Ship(length=250)
```

This sets ship.length to 250.

In most cases, the setup method is preferred, however. This method is called after ALL initialization code of Component is executed.

```
class Ship(sim.Component):
    def setup(self, length):
        self.length = length

ship = Ship(length=250)
```

Now, ship.length will be 250.

Note that setup gets all arguments and keyword arguments, that are not 'consumed' by __init__ and/or the process call.

Only in very specific cases, __init__ will be necessary.

Note that the setup code can be used for data components as well.

3.1 Process interaction

A component may be in one of the following states:

- data
- current
- scheduled
- passive
- requesting
- waiting
- standby
- interrupted

The scheme below shows how components can go from state to state.

from/to	data	current	scheduled	passive	requesting	waiting	standby	interrupted
data		activate[1]	activate					
current	process end		yield hold	yield passivate	yield request	yield wait	yield standby	
.	yield cancel		yield activate					
scheduled	cancel	next event	hold	passivate	request	wait	standby	interrupt
.			activate					
passive	cancel	activate[1]	activate		request	wait	standby	interrupt
.			hold[2]					
requesting	cancel	claim honor	activate[3]	passivate	request	wait	standby	interrupt
.		time out			activate[4]			
waiting	cancel	wait honor	activate[5]	passivate	wait	wait	standby	interrupt
.		timeout				activate[6]		
standby	cancel	next event	activate	passivate	request	wait		interrupt
interrupted	cancel		resume[7]	resume[7]	resume[7]	resume[7]	resume[7]	interrupt[8]
.			activate	passivate	request	wait	standby	

[1] via scheduled [2] not recommended [3] with keep_request=False (default) [4] with keep_request=True. This allows to set a new time out [5] with keep_wait=False (default) [6] with keep_wait=True. This allows to set a new time out [7] state at time of interrupt [8] increases the interrupt_level

3.1.1 Creation of a component

Although it is possible to create a component directly with *x=sim.Component()*, this makes it very hard to make that component into an active component, because there's no process method. So, nearly always we define a class based on sim.Component

```
def Car(sim.Component):
    def process(self):
        ...
```

If we then say `car=Car()`, a component is created and it activated from process. This process is nearly always, but not necessarily a generator method (i.e. it has at least one yield (or yield from) statement.

The result is that car is put on the future event list (for time now) and when it's its turn, the component becomes current.

It is also possible to set a time at which the component (car) becomes active, like *car=Car(at=10)*.

And instead of starting at process, the component may be initialized to start at another (generator) method, like `car=Car(process='wash')`.

And, finally, if there is a process method, you can disable the automatic activation (i.e. make it a data component) , by specifying `process=''`.

If there is no process method, and process= is not given, the component will be a data component.

3.1.2 activate

Activate is the way to turn a data component into a live component. If you do not specify a process, the (usually generator) function process is assumed. So you can say

```
car0 = Car(process='')  # data component
car0.activate()  # activate @ process if exists, otherwise error
car1 = Car(process='')  # data component
car1.activate(process='wash')  # activate @ wash
```

- If the component to be activated is current, always use yield self.activate. The effect is that the component becomes scheduled, thus this is essentially equivalent to the preferred hold method.
- If the component to be activated is passive, the component will be activated at the specified time.
- If the component to be activated is scheduled, the component will get a new scheduled time.
- If the component to be activated is requesting, the request will be terminated, the attribute failed set and the component will become scheduled. If keep_request=True is specified, only the fail_at will be updated and the component will stay requesting.
- If the component to be activated is waiting, the wait will be terminated, the attribute failed set and the component will become scheduled. If keep_wait=True is specified, only the fail_at will be updated and the component will stay waiting.
- If the component to be activated is standby, the component will get a new scheduled time and become scheduled.
- If the component is interrupted, the component will be activated at the specified time.

3.1.3 hold

Hold is the way to make a, usually current, component scheduled.

- If the component to be held is current, the component becomes scheduled for the specified time. Always use yield self.hold() is this case.
- If the component to be held is passive, the component becomes scheduled for the specified time.
- If the component to be held is scheduled, the component will be rescheduled for the specified time, thus essentially the same as activate.
- If the component to be held is standby, the component becomes scheduled for the specified time.
- If the component to be activated is requesting, the request will be terminated, the attribute failed set and the component will become scheduled. It is recommended to use the more versatile activate method.

- If the component to be activated is waiting, the wait will be terminated, the attribute failed set and the component will become scheduled. It is recommended to use the more versatile activate method.
- If the component is interrupted, the component will be activated at the specified time.

3.1.4 passivate

Passivate is the way to make a, usually current, component passive. This isessentially the same as scheduling for time=inf.

- If the component to be passivated is current, the component becomes passive. Always use yield self.passivate() is this case.
- If the component to be passivated is passive, the component remains passive.
- If the component to be passivated is scheduled, the component becomes passive.
- If the component to be held is standby, the component becomes passive.
- If the component to be activated is requesting, the request will be terminated, the attribute failed set and the component becomes passive. It is recommended to use the more versatile activate method.
- If the component to be activated is waiting, the wait will be terminated, the attribute failed set and the component becomes passive. It is recommended to use the more versatile activate method.
- If the component is interrupted, the component becomes passive.

3.1.5 cancel

Cancel has the effect that the component becomes a data component.

- If the component to be cancelled is current, always use yield self.cancel().
- If the component to be cancelled is passive, scheduled, interrupted or standby, the component becomes a data component.
- If the component to be cancelled is requesting, the request will be terminated, the attribute failed set and the component becomes a data component.
- If the component to be cancelled is waiting, the wait will be terminated, the attribute failed set and the component becomes a data component.

3.1.6 standby

Standby has the effect that the component will be triggered on the next simulation event.

- If the component is current, use always yield self.standby()
- Although theoretically possible, it is not recommended to use standby for non current components.

3.1.7 request

Request has the effect that the component will check whether the requested quantity from a resource is available. It is possible to check for multiple availability of a certain quantity from several resources.

Instead of checking for all of number of resources, it is also possible to check for any of a number of resources, by setting the oneof parameter to True.

By default, there is no limit on the time to wait for the resource(s) to become available. But, it is possible to set a time with fail_at at which the condition has to be met. If that failed, the component becomes current at the given point of time. The code should then check whether the request had failed. That can be checked with the Component.failed() method.

If the component is canceled, activated, passivated, interrupted or held the failed flag will be set as well.

- If the component is current, always use yield self.request()
- Although theoretically possible it is not recommended to use request for non current components.

3.1.8 wait

Wait has the effect that the component will check whether the value of a state meets a given condition. available. It is possible to check for multiple states. By default, there is no limit on the time to wait for the condition(s) to be met. But, it is possible to set a time with fail_at at which the condition has to be met. If that failed, the component becomes current at the given point of time. The code should then check whether the wait had failed. That can be checked with the Component.failed() method.

If the component is canceled, activated, passivated, interrupted or held the failed flag will be set as well.

- If the component is current, use always yield self.wait()
- Although theoretically possible it is not recommended to use wait for non current components.

3.1.9 interrupt

With interrupt components that are not current or data can be temporarily be interrupted. Once a resume is called for the component, the component will continue (for scheduled with the remaining time, for waiting or requesting possibly with the remaining fail_at duration).

3.2 Usage of process interaction methods within a function or method

There is a way to put process interaction statement in another function or method. This requires a slightly different way than just calling the method.

As an example, let's assume that we want a method that holds a component for a number of minutes and that the time unit is actually seconds. So we need a method to wait 60 times the given parameter

We start with a not so elegant solution:

```
class X(sim.Component):
    def process(self):
        yield self.hold(60 * 2)
        yield self.hold(60 * 5)
```

Now we just addd a method hold_minutes:

```
def hold_minutes(self, minutes):
    yield self.hold(60 * minutes)
```

Direct calling hold_minutes is not possible. Instead we have to say:

```
class X(sim.Component):
   def hold_minutes(self, minutes):
        yield self.hold(60 * minutes)

   def process(self):
        yield from self.hold_minutes(2)
        yield from self.hold_minutes(5)
```

All process interaction statements including passivate, request and wait can be used that way!

So remember if the method contains a yield statement (technically speaking that's a generator method), it should be called with `yield from`.

3.3 Using priority and urgent to control order of execution

All process interaction methods supports a priority and urgent parameter:

With priority it is possible to sort a component before or after other components, scheduled for the same time. Note that the urgent parameters only applies to components scheduled with the same time and same priority.

The priority is 0 by default.

This is particularly useful for race conditions. It is possible to change the priority of a component by cancelling it prior to activating it with another priority.

The priority can be accessed with the new Component.scheduled_priority() method.

3.4 Status of a component

The status of a component can be any of:

- sim.data = “data”
- sim.current = “current”
- sim.standby = “standby”
- sim.passive = “passive”
- sim.interrupted = “interrupted”
- sim.scheduled = “scheduled”
- sim.requesting = “requesting”
- sim.waiting = “waiting”

The status is automatically tracked in the status level monitor. Thus it possible to check how long a component has been in passive state with

```
passive_duration = component.status.value_duration("passive")
```

And it is possible to print a histogram with all the statuses a component has been in with

```
component.status.print_histogram(values=True)
```

3.5 Handling negative durations

When sampling from a distribution (particularly a normal distribution), a negative duration may be returned

```
yield self.hold(sim.Normal(5, 3))
```

Such negative values can’t be used to hold a period, as salabim can’t go back in time and thus raise an exception.

There are a couple of ways to prevent a negative value to be returned

```
yield self.hold(sim.Normal(5, 3).bounded_sample(lowerbound=0))

yield self.hold(sim.Bounded(sim.Normal(5, 3), lower_bound=0))
```

In both cases, if negative value is sampled, salabim will resample until a value >=0 is found.

Alternatively

```
yield self.hold(sim.Map(sim.Normal(5, 3), lambda x: 0 if x<0 else x)
```

This will map negative values to 0.

Finally, we can do

```
yield self.hold(Normal(5, 3), cap_now=True)
```

The latter version just uses 0 in case a negative value is given. Be careful with this as it might hide model mistakes.

The *cap_now* parameter is available for each method that sets a scheduled time.

The *cap_now* functionality can also be enabled globally

```
sim.default_cap_now(True)
```

Or for a certain block

```
with sim.cap_now():
    yield self.hold(Normal(5, 3))
    yield self.hold(Normal(12, 7))
```

CHAPTER

FOUR

COMPONENTGENERATOR

With ComponentGenerator components can be generated according to a given inter arrival time (distribution) or a random spread over a given interval.

Examples

```
sim.ComponentGenerator(Car, iat=sim.Exponential(2))
# generates Cars according to a Poisson arrival

sim.ComponentGenerator(Car, iat=sim.Exponential(2), at=10, till=30, number=10)
# generates maximum 10 Cars according to a Poisson arrival from t=10 to t=30

sim.ComponentGenerator(Car, iat=sim.Exponential(2), at=10, till=30, number=10, force_at=True)
# generates maximum 10 Cars according to a Poisson arrival from t=10 to t=30, first arrival at t=10

sim.ComponentGenerator(sim.Pdf((Car, 0.7, Bus, 0.3)), iat=sim.Uniform(20,40), number=20)
# generates maximum 20 vehicles (70% Car, 30% Bus) according to a uniform inter arrival time

sim.ComponentGenerator(Car, duration=100, n=20)
# generates exactly 20 Cars, random spread over t=now and t=now+100

sim.ComponentGenerator(Car, duration=100, n=20, force_at=True, force_till=True)
# generates exactly 20 Cars, random spread over t=now and t=now+100, with arrivals at t=now and t=now+100
```

ComponentGenerator is a subclass of Component and therefore has all the normal properties and methods of an ordinary component, altough it is not recommended to use any of the process methods, apart from cancel.

It is possible to ‘propagate’ parameters to a generated component. This holds for the standard salabim parameters, like name, urgent and priority but also for user class defined parameters.

E.g.

```
sim.ComponentGenerator(Car, iat=sim.Exponential(2), color='red')
# generates Cars with color=color according to a Poisson arrival

sim.ComponentGenerator(Car, iat=sim.Exponential(2), name='ford.')
# generates Cars according to a Poisson arrival with name ford.0, ford.1
```

The component_class can also be a callable without paramters, most like a distribution.

So we can say

```
sim.ComponentGenerator(sim.Pdf((Car, Bus, Truck), (50, 30, 20)), iat=sim.Exponential(2))
# generates 50% Cars, 30% Buses and 20% Trucks according to a Poisson arrival
```

CHAPTER

FIVE

QUEUE

Salabim has a class Queue for queue handling of components. The advantage over the standard list and deque are:

- double linked, resulting in easy and efficient insertion and deletion at any place
- builtin data collection and statistics
- priority sorting

Salabim uses queues internally for resources and states as well.

Definition of a queue is simple:

```
waitingline=sim.Queue('waitingline')
```

The name of a queue can retrieved with `q.name()`.

There is a set of methods for components to enter and leave a queue and retrieval:

Component	Queue	Description
c.enter(q)	q.add(c) or q.append(c)	c enters q at the tail
c.enter_to_head(q)	q.add_at_head(c)	c enters q at the head
c.enter_in_front(q, c1)	q.add_in_front_of(c, c1)	c enters q in front of c1
c.enter_behind(q, c1)	q.add_behind(c, c1)‘	c enters q behind c1
c.enter_sorted(q, p)	q.add_sorted(c, p)	c enters q according to priority p
c.leave(q)	q.remove(c) q.insert(c,i) q.pop() q.pop(i) q.head() or q[0] q.tail() or q[-1] q.index(c) q.component_with_name(n)	c leaves q insert c just before the i-th component in q removes head of q and returns it removes i-th component in q and returns it returns head of q returns tail of q returns the position of c in q returns the component with name n in q
c.successor(q)	q.successor(c)	successor of c in q
c.predecessor(q)	q.predecessor(c)	predecessor of c in q
c.count(q)	q.count(c)	returns 1 if c in q, 0 otherwise
c.queues()		returns a set with all queues where c is in
c.count()	returns number of queues c is in	

Queue is a standard ABC class, which means that the following methods are supported:

- `len(q)` to retrieve the length of a queue, alternatively via the level monitor with `q.length()`
- `c in q` to check whether a component is in a queue
- `for c in q:` to traverse a queue (Note that it is even possible to remove and add components in the for body).
- `reversed(q)` for the components in the queue in reverse order
- slicing is supported, so it is possible to get the 2nd, 3rd and 4th component in a queue with `q[1:4]` or `q[::-1]` for all elements in reverse order.
- `del q[i]` removes the i'th component. Also slicing is supported, so e.g. to delete the last three elements from queue, `del q[-1:-4:-1]`
- `q.append(c)` is equivalent to `q.add(c)`

It is possible to do a number of operations that work on the queues:

- `q.intersection(q1)` or `q & q1` returns a new queue with components that are both in q and q1
- `q.difference(q1)` or `` q - q1` `` returns a new queue with components that are in q1 but not in q2
- `q.union(q1)` or `q | q1` or `q + q1` returns a new queue with components that are in q or q1

- `q.symmetric_difference(q)` or `q ^ q1` returns a queue with components that are in q or q1, but not both
- `q.clear()` empties a queue
- `q.copy()` copies all components in q to a new queue. The queue q is untouched.
- `q.move()` copies all components in q to a new queue. The queue q is emptied.
- `q.extend(q1)` extends the q with elements in q1, that are not yet in q

Note that it is possible to rename a queue (particularly those created with +, -, ^, | or sum) with the rename() method:

```
(q0 | q1 | q2 | q3).rename('q0 - q3).print_info()
```

Salabim keeps track of the enter time in a queue: `c.enter_time (q)`

Unless disabled explicitly, the length of the queue and length of stay of components are monitored in `q.length` and `q.length_of_stay`. It is possible to obtain a number of statistics on these monitors (cf. Monitor).

With `q.print_statistics()` the key statistics of these two monitors are printed.

E.g.:

```
-------------------------------------------- -------------- ------------ ------------ ------------
Length of waitingline                        duration          50000        48499.381     1500.619
                                             mean                  8.427        8.687
                                             std.deviation         4.852        4.691

                                             minimum               0            1
                                             median                9           10
                                             90% percentile       14           14
                                             95% percentile       16           16
                                             maximum              21           21

Length of stay in waitingline                entries            4995         4933           62
                                             mean                 84.345       85.405
                                             std.deviation        48.309       47.672

                                             minimum               0            0.006
                                             median               94.843       95.411
                                             90% percentile      142.751      142.975
                                             95% percentile      157.467      157.611
                                             maximum             202.153      202.153
```

The arrival rate and departure rate (number of arrivals, departures per time unit) can be found with:

- q.arrival_rate()
- q.departur_rate()

With `q.print_info()` a summary of the contents of a queue can be printed.

E.g.

```
Queue 0x20e116153c8
  name=waitingline
  component(s):
    customer.4995        enter_time 49978.472 priority=0
    customer.4996        enter_time 49991.298 priority=0
```

CHAPTER

SIX

RESOURCE

Resources are a powerful way of process interaction.

A resource has always a capacity (which can be zero and even negative). This capacity will be specified at time of creation, but may change over time. There are two of types resources:

- standard resources, where each claim is associated with a component (the claimer). It is not necessary that the claimed quantities are integer.
- anonymous resources, where only the claimed quantity is registered. This is most useful for dealing with levels, lengths, etc.

Resources are defined like

```
clerks = Resource('clerks', capacity=3)
```

And then a component can request a clerk

```
yield self.request(clerks)  # request 1 from clerks
```

It is also possible to request for more resources at once

```
yield self.request(clerks,(assistance,2))  # request 1 from clerks AND 2 from assistance
```

AResources have a queue `requesters` containing all components trying to claim from the resource. And a queue `claimers` containing all components claiming from the resource (not for anonymous resources).

It is possible to release a quantity from a resource with c.release(), e.g.

```
self.release(r)  # releases all claimed quantity from r
self.release((r,2))  # release quantity 2 from r
```

Alternatively, it is possible to release from a resource directly, e.g.

```
r.release()  # releases the total quantity from all claiming components
r.release(10)  # releases 10 from the resource; only valid for anonymous resources
```

After a release, all requesting components will be checked whether their claim can be honored.

Resources have a number monitors:

- claimers().length
- claimers().length_of_stay
- requesters().length
- requesters().length_of_stay
- claimed_quantity
- available_quantity
- capacity
- occupancy (=claimed_quantity / capacity)

By default, all monitors are enabled.

With `r.print_statistics()` the key statistics of these all monitors are printed.

E.g.:

```
Statistics of clerk at     50000.000
                                                                 all    excl.zero         zero
-------------------------------------------- -------------- ------------ ------------ ------------
Length of requesters of clerk                duration          50000        48499.381     1500.619
                                             mean                  8.427        8.687
                                             std.deviation         4.852        4.691

                                             minimum               0            1
                                             median                9           10
                                             90% percentile       14           14
                                             95% percentile       16           16
                                             maximum              21           21

Length of stay in requesters of clerk        entries            4995         4933           62
```

```
                                             mean                84.345       85.405
                                             std.deviation       48.309       47.672

                                             minimum              0            0.006
                                             median              94.843       95.411
                                             90% percentile     142.751      142.975
                                             95% percentile     157.467      157.611
                                             maximum            202.153      202.153

Length of claimers of clerk                  duration         50000        50000             0
                                             mean                 2.996        2.996
                                             std.deviation        0.068        0.068

                                             minimum              1            1
                                             median               3            3
                                             90% percentile       3            3
                                             95% percentile       3            3
                                             maximum              3            3

Length of stay in claimers of clerk          entries           4992         4992             0
                                             mean                30           30
                                             std.deviation        0.000        0.000

                                             minimum             30.000       30.000
                                             median              30           30
                                             90% percentile      30           30
                                             95% percentile      30           30
                                             maximum             30.000       30.000

Capacity of clerk                            duration         50000        50000             0
                                             mean                 3            3
                                             std.deviation        0            0

                                             minimum              3            3
                                             median               3            3
                                             90% percentile       3            3
                                             95% percentile       3            3
                                             maximum              3            3

Available quantity of clerk                  duration         50000          187.145    49812.855
```

```
                                             mean                   0.004        1.078
                                             std.deviation          0.068        0.268

                                             minimum                0            1
                                             median                 0            1
                                             90% percentile         0            1
                                             95% percentile         0            2
                                             maximum                2            2

Claimed quantity of clerk                    duration           50000        50000            0
                                             mean                   2.996        2.996
                                             std.deviation          0.068        0.068

                                             minimum                1            1
                                             median                 3            3
                                             90% percentile         3            3
                                             95% percentile         3            3
                                             maximum                3            3

Occupancy of clerks                          duration           50000        50000            0
                                             mean                   0.999        0.999
                                             std.deviation          0.023        0.023

                                             minimum                0.333        0.333
                                             median                 1            1
                                             90% percentile         1            1
                                             95% percentile         1            1
                                             maximum                1            1
```

With `r.print_info()` a summary of the contents of the queues can be printed.

E.g.

```
Resource 0x112e8f0b8
  name=clerk
  capacity=3
  requesting component(s):
    customer.4995        quantity=1
    customer.4996        quantity=1
  claimed_quantity=3
  claimed by:
    customer.4992        quantity=1
    customer.4993        quantity=1
    customer.4994        quantity=1
```

The capacity may be changed with `r.set_capacity(x)`. Note that this may lead to requesting components to be honored.

Querying of the capacity, claimed quantity, available quantity and occupancy can be done via the label monitors: `r.capacity()`, `r.claimed_quantity()`, `r.available_quantity()` and `r.occupancy()`

If the capacity of a resource is constant, which is very common, the mean occupancy can be found with

```
r.occupancy.mean()
```

When the capacity changes over time, it is recommended to use

```
occupancy = r.claimed_quantity.mean() / r.capacity.mean()
```

to obtain the mean occupancy.

Note that the occupancy is set to 0 if the capacity of the resource is <= 0.

6.1 Additional methods for anonymous resources

For anonymous resources, it may be not allowed to exceed the capacity and have a component wait for enough (claimed) capacity to be available. That may be accomplished by using a negative quantity in the `self.request` call.

Alternatively, it possible to use the `Component.put` method, where quantities of anonymous resources are negated. For symmetry reasons, salabim also offers the `Component.get()` method, which is behaves exactly like `Component.request`.

The model below illustrates the use of get and put.

```python
#  Gas station.py
import salabim as sim

#  based on SimPy example model

GAS_STATION_SIZE = 200.0  # liters
THRESHOLD = 25.0  # Threshold for calling the tank truck (in %)
FUEL_TANK_SIZE = 50.0  # liters
```

```
# Min/max levels of fuel tanks (in liters)
FUEL_TANK_LEVEL = sim.Uniform(5, 25)
REFUELING_SPEED = 2.0  # liters / second
TANK_TRUCK_TIME = 300.0  # Seconds it takes the tank truck to arrive
T_INTER = sim.Uniform(10, 100)  # Create a car every [min, max] seconds
SIM_TIME = 200000  # Simulation time in seconds


class Car(sim.Component):
    """
    A car arrives at the gas station for refueling.

    It requests one of the gas station's fuel pumps and tries to get the
    desired amount of gas from it. If the stations reservoir is
    depleted, the car has to wait for the tank truck to arrive.

    """

    def process(self):
        fuel_tank_level = int(FUEL_TANK_LEVEL.sample())
        yield self.request(gas_station)
        liters_required = FUEL_TANK_SIZE - fuel_tank_level
        if (fuel_pump.available_quantity() - liters_required) / fuel_pump.capacity() * 100 < THRESHOLD:
            TankTruck()
        yield self.get((fuel_pump, liters_required))
        yield self.hold(liters_required / REFUELING_SPEED)


class TankTruck(sim.Component):
    def process(self):
        yield self.hold(TANK_TRUCK_TIME)
        amount = fuel_pump.claimed_quantity()
        yield self.put((fuel_pump, amount))


class CarGenerator(sim.Component):
    """
    Generate new cars that arrive at the gas station.
    """

    def process(self):
        while True:
            yield self.hold(T_INTER.sample())
            Car()


# Setup and start the simulation
env = sim.Environment(trace=False)
print("Gas Station refuelling")

# Create environment and start processes
gas_station = sim.Resource("gas_station", 2)
fuel_pump = sim.Resource("fuel_pump", capacity=GAS_STATION_SIZE, anonymous=True)
tank_truck = TankTruck()
CarGenerator()

env.run(SIM_TIME)

fuel_pump.capacity.print_histogram()
fuel_pump.claimed_quantity.print_histogram()
fuel_pump.available_quantity.print_histogram()


gas_station.requesters().length.print_histogram()
gas_station.requesters().length_of_stay.print_histogram(30, 0, 10)
```

CHAPTER

SEVEN

STATE

States together with the Component.wait() method provide a powerful way of process interaction.

A state will have a certain value at a given time. In its simplest form a component can then wait for a specific value of a state. Once that value is reached, the component will be resumed.

Definition is simple, like `dooropen=sim.State('dooropen')`. The default initial value is False, meaning the door is closed.

Now we can say

```
dooropen.set()
```

to open the door.

If we want a person to wait for an open door, we could say

```
yield self.wait(dooropen)
```

If we just want at most one person to enter, we say `dooropen.trigger(max=1)`.

We can obtain the current value by just calling the state, like in

```
print('door is ',('open' if dooropen() else 'closed'))
```

Alternatively, we can get the current value with the get method

```
print('door is ',('open' if dooropen.get() else 'closed'))
```

The value of a state is automatically monitored in the state.value level monitor.

All components waiting for a state are in a salabim queue, called waiters().

States can be used also for non values other than bool type. E.g.

```
light=sim.State('light', value='red')
...
light.state.set('green')
```

Or define a int/float state

```
level=sim.State('level', value=0)
...
level.set(level()+10)
```

States have a number of monitors:

- value, where all the values are collected over time
- waiters().length
- waiters().length_of_stay

7.1 Process interaction with wait()

A component can wait for a state to get a certain value. In its most simple form

```
yield self.wait(dooropen)
```

Once the dooropen state is True, the component will continue.

As with request() it is possible to set a timeout with fail_at or fail_delay

```
yield self.wait(dooropen, fail_delay=10)
if self.failed():
    print('impatient ...')
```

In the above example we tested for a state to be True.

There are three ways to test for a value:

7.1.1 Scalar testing

It is possible to test for a certain value

```
yield self.wait((light, 'green'))
```

Or more states at once

```
yield self.wait((light, 'green'), night)  # honored as soon as light is green OR it's night
yield self.wait((light, 'green'), (light, 'yellow'))  # honored as soon is light is green OR yellow
```

It is also possible to wait for all conditions to be satisfied, by adding `all=True`:

```
yield self.wait((light,'green'), enginerunning, all=True)  # honored as soon as light is green AND engine is running
```

7.1.2 Evaluation testing

Here, we use a string containing an expression that can evaluate to True or False. This is done by specifying at least one `$` in the test-string. This `$` will be replaced at run time by `state.value()`, where state is the state under test. Here are some examples

```
yield self.wait((light, '$ in ("green","yellow")'))
    # if at run time light.value() is 'green', test for eval(state.value() in ("green,"yellow")) ==> True
yield self.wait((level, '$ < 30'))
    # if at run time level.value() is 50, test for eval(state.value() < 30) ==> False
```

During the evaluation, `self` refers to the component under test and `state` to the state under test. E.g.

```
self.limit = 30
yield self.wait((level, 'self.limit >= $'))
    # if at run time level.value() is 10, test for eval(self.limit >= state.get()) ==> True, so honored
```

7.1.3 Function testing

This is a more complicated but also more versatile way of specifying the honor-condition. In that case, a function is required to specify the condition. The function needs to accept three arguments:

- x = state.get()
- component component under test
- state under test

E.g.:

```
yield self.wait((light, lambda x, component, state x: in ('green', 'yellow'))
    # x is light.get()
yield self.wait((level, lambda x, *_: x >= 30))
    # x is level.get(), other two parameters are 'dummied'
```

And, of course, it is possible to define a function

```
def levelreached(value, component, state):
    return value < component.limit

...

self.limit = 30
yield self.wait((level, levelreached))
```

7.1.4 Combination of testing methods

It is possible to mix scalar, evaluation and function testing. And it's also possible to specify all=True in any case.

CHAPTER

EIGHT

MONITOR

Monitors are a way to collect data from the simulation. They are automatically collected for resources, queues and states. On top of that the user can define its own monitors.

Monitors can be used to get statistics and as a feed for graphical tools, like matplotlib.

There are two types of monitors:

- level monitors Level monitors are useful to collect data about a variable that keeps its value over a certain length of time, such as the length of a queue orcthe colour of a traffic light.
- non level monitors Non level monitors are useful to collect data about a values that occur just once. Examples, are the length of stay in a queue and the number of processing steps of a part.

For both types, the time is always collected, along with the value.

Non level monitors can be weighted, if required.

8.1 Non level monitor

Non level monitors collects values which do not refelect a level, e.g. the processing time of a part.

We define the monitor with `processingtime = sim.Monitor('processingtime')` and then collect values by `processingtime.tally(this_duration)`

By default, the collected values are stored in a list. Alternatively, it is possible to store the values in an array of one of the following types:

type	stored as	lowerbound	upperbound	number of bytes
‘any’	list	N/A	N/A	depends on data
‘bool’	integer	False	True	1
‘int8’	integer	-128	127	1
‘uint8’	integer	0	255	1
‘int16’	integer	-32768	32767	2
‘uint16’	integer	0	65535	2
‘int32’	integer	2147483648	2147483647	4
‘uint32’	integer	0	4294967295	4
‘int64’	integer	-9223372036854775808	9223372036854775807	8
‘uint64’	integer	0	18446744073709551615	8
‘float’	float	-inf	inf	8

Monitoring with arrays takes up less space. Particularly when tallying a large number of values, this is strongly advised.

Note that if non numeric values are stored (only possible with the default setting (‘any’)), a tallied value is converted, if required, to a numeric value if possible, or 0 if not.

It is possible to use monitors with weighted data. In that case, just add a second parameter to tally, which defaults to 1. All statistics will take the weights into account.

There is set of statistical data available, which will be all weighed according to the tallied weights (1 by default):

- number_of_entries
- number_of_entries_zero
- weight
- weight_zero
- mean
- std
- minimum
- median
- maximum
- percentile
- bin_number_of_entries (number of entries between two given values)
- bin_weight (total weight of entries between two given values)

- value_number_of_entries (number of entries equal to a given value or set of values)
- value_weight (total weight of entries equal to a given value or set of values)

For all these statistics, it is possible to exclude zero entries, e.g. `m.mean(ex0=True)` returns the mean, excluding zero entries.

Besides, it is possible to get all collected values as an array with x(). In the case of 'any' monitors, the values might have to be converted. By specifying `force_numeric=False` the collected values will be returned as stored.

With the monitor method, the monitor can be enbled or disabled. Note that a tally is just ignored when the monitor is disabled.

Also, the current monitor status (enabled/disabled) can be retrieved.

```
proctime.monitor(False)  # disable monitoring
proctime.monitor(True)  # enable monitoring
if proctime.monitor():
    print('proctime is enabled')
```

Calling m.reset() will clear all tallied values.

The statistics of a monitor can be printed with `print_statistics()`. E.g: `waitingline.length_of_stay.print_statistics()`:

```
Statistics of Length of stay in waitingline at     50000
                        all    excl.zero         zero
-------------- ------------ ------------ ------------
entries            4995         4933           62
mean                 84.345       85.405
std.deviation        48.309       47.672

minimum               0            0.006
median               94.843       95.411
90% percentile      142.751      142.975
95% percentile      157.467      157.611
maximum             202.153      202.153
```

And, a histogram can be printed with `print_histogram()`. E.g. `waitingline.length_of_stay.print_histogram(30, 0, 10)`:

```
Histogram of Length of stay in waitingline
                        all    excl.zero         zero
-------------- ------------ ------------ ------------
entries            4995         4933           62
mean                 84.345       85.405
std.deviation        48.309       47.672
```

```
minimum               0            0.006
median               94.843       95.411
90% percentile      142.751      142.975
95% percentile      157.467      157.611
maximum             202.153      202.153

           <=       entries     %  cum%
        0            62       1.2   1.2 |
       10           169       3.4   4.6 ** |
       20           284       5.7  10.3 ****    |
       30           424       8.5  18.8 ******         |
       40           372       7.4  26.2 *****               |
       50           296       5.9  32.2 ****                     |
       60           231       4.6  36.8 ***                          |
       70           192       3.8  40.6 ***                             |
       80           188       3.8  44.4 ***                                |
       90           136       2.7  47.1 **                                   |
      100           352       7.0  54.2 *****                                      |
      110           491       9.8  64.0 *******                                            |
      120           414       8.3  72.3 ******                                                   |
      130           467       9.3  81.6 *******                                                          |
      140           351       7.0  88.7 *****                                                                 |
      150           224       4.5  93.2 ***                                                                       |
      160           127       2.5  95.7 **                                                                          |
      170            67       1.3  97.0 *                                                                            |
      180            59       1.2  98.2                                                                               |
      190            61       1.2  99.4                                                                                |
      200            24       0.5  99.9                                                                                |
      210             4       0.1 100                                                                                   |
      220             0       0   100                                                                                   |
      230             0       0   100                                                                                   |
      240             0       0   100                                                                                   |
      250             0       0   100                                                                                   |
      260             0       0   100                                                                                   |
      270             0       0   100                                                                                   |
      280             0       0   100                                                                                   |
      290             0       0   100                                                                                   |
      300             0       0   100                                                                                   |
          inf         0       0   100
```

If neither number_of_bins, nor lowerbound nor bin_width are specified, the histogram will be autoscaled.

Histograms can be printed with their values, instead of bins. This is particularly useful for non numeric tallied values, such as names:

```
import salabim as sim

env = sim.Environment()

monitor_names= sim.Monitor(name='names')
for _ in range(10000):
    name = sim.Pdf(('John', 30, 'Peter', 20, 'Mike', 20, 'Andrew', 20, 'Ruud', 5, 'Jan', 5)).sample()
    monitor_names.tally(name)

monitor_names.print_histogram(values=True)
```

The ouput of this:

```
Histogram of names
entries          10000

value                entries
Andrew                  2031( 20.3%) ****************
Jan                      495(  5.0%) ***
John                    2961( 29.6%) ***********************
Mike                    1989( 19.9%) ***************
Peter                   2048( 20.5%) ****************
Ruud                     476(  4.8%) ***
```

It is also possible to specify the values to be shown

```
import salabim as sim

env = sim.Environment()

monitor_names= sim.Monitor(name='names')
for _ in range(10000):
    name = sim.Pdf(('John', 30, 'Peter', 20, 'Mike', 20, 'Andrew', 20, 'Ruud', 5, 'Jan', 5)).sample()
    monitor_names.tally(name)

monitor_names.print_histogram(values=('John', 'Andrew', 'Ruud', 'Fred'))
```

The output of this:

```
Histogram of names
entries          10000

value                  entries     %
John                      2961  29.6 ***********************
Andrew                    2031  20.3 ****************
Ruud                       476   4.8 ***
Fred                         0   0
<rest>                    4532  45.3 ************************************
```

It is also possible to sort the histogram on the weight (or number of entries) of the value

```
import salabim as sim

env = sim.Environment()

monitor_names= sim.Monitor(name='names')
for _ in range(10000):
    name = sim.Pdf(('John', 30, 'Peter', 20, 'Mike', 20, 'Andrew', 20, 'Ruud', 5, 'Jan', 5)).sample()
    monitor_names.tally(name)

monitor_names.print_histogram(values=True, sort_on_weight=True)
```

The output of this:

```
Histogram of names
entries          10000

value                  entries     %
John                      2961  29.6 ***********************
Peter                     2048  20.5 ****************
Andrew                    2031  20.3 ****************
Mike                      1989  19.9 ***************
Jan                        495   5.0 ***
Ruud                       476   4.8 ***
```

8.2 Level monitor

Level monitors tally levels along with the current (simulation) time. e.g. the number of parts a machine is working on.

A level monitor is defined by specifying `level=True` in the initialization of Monitor, e.g.

```
working_on_parts = sim.Monitor(name='working_on_parts', level=True, initial_tally=0)
```

By default, the collected x-values are stored in a list. Alternatively, it is possible to store the x-values in an array of one of the following types:

type	stored as	lowerbound	upperbound	number of bytes	do not tally (=off)
'any'	list	N/A	N/A	depends on data	N/A'
'bool'	integer	False	True	1	255
'int8'	integer	-127	127	1	-128
'uint8'	integer	0	254	1	255
'int16'	integer	-32767	32767	2	-32768
'uint16'	integer	0	65534	2	65535
'int32'	integer	2147483647	2147483647	4	2147483648
'uint32'	integer	0	4294967294	4	4294967295
'int64'	integer	-9223372036854775807	9223372036854775807	8	-9223372036854775808
'uint64'	integer	0	18446744073709551614	8	18446744073709551615
'float'	float	-inf	inf	8	-inf

Monitoring with arrays takes up less space. Particularly when tallying a large number of values, this is strongly advised.

Note that if non numeric x-values are stored (only possible with the default setting ('any')), the tallied values are converted, if required, to a numeric value if possible, or 0 if not.

During the simulation run, it is possible to retrieve the last tallied value (which represents the 'current' value) by calling Monitor.get(). It's also possible to directly call the level monitor to get the current value, e.g.

```
mylevel = sim.Monitor('level', level=True, initial_tally=0)
...
mylevel.tally(10)
yield seld.hold(1)
print(mylevel())  # will print 10
```

For the same reason, the standard length monitor of a queue can be used to get the current length of a queue: `q.length()` although the more Pythonic `len(q)` is prefered.

When Monitor.get() is called with a time parameter or a direct call with a time parameter, the value at that time will be returned

```
print (mylevel.get(4))  # will print the value at time 4
print (mylevel(4))  # will print the value at time 4
```

There is set of statistical data available, which are all weighted with their duration:

- duration
- duration_zero (time that the value was zero)
- mean
- std
- minimum
- median
- maximum
- percentile
- bin_duration (total duration of entries between two given values)
- value_duration (total duration of entries equal to a given value or set of values)

For all these statistics, it is possible to exclude zero entries, e.g. `m.mean(ex0=True)` returns the mean, excluding zero entries.

The individual x-values and their duration can be retrieved xduration(). By default, the x-values will be returned as an array, even if the type is 'any'. In case the type is 'any' (stored as a list), the tallied x-values will be converted to a numeric value or 0 if that's not possible. By specifying `force_numeric=False` the collected x-values will be returned as stored.

The individual x-values and the associated timestamps can be retrieved with xt() or tx(). By default, the x-values will be returned as an array, even if the type is 'any'. In case the type is 'any' (stored as a list), the tallied x-values will be converted to a numeric value or 0 if that's not possible. By specifying `force_numeric=False` the collected x-values will be returned as stored.

When monitoring is disabled, an off value (see table above) will be tallied. All statistics will ignore the periods from this off to a non-off value. This also holds for the xduration() method, but NOT for xt() and tx(). Thus, the x-arrays of xduration() are not necessarily the same as the x-arrays in xt() and tx(). This is the reason why there's no x() or t() method. It is easy to get just the x-array with xduration()[0] or xt()[0].

It is important that a user *never* tallies an off value! Instead use Monitor.monitor(False)

With the monitor method, a level monitor can be enbled or disabled.

Also, the current monitor status (enabled/disabled) can be retrieved.

```
mylevel.monitor(False)  # disable monitoring
mylevel.monitor(True)  # enable monitoring
if mylevel.monitor():
    print('level is enabled')
```

It is strongly advised to keep tallying even when monitoring is off, in order to be able to access the current value at any time. The values tallied when monitoring is off are not stored.

Calling m.reset() will clear all tallied values and timestamps.

The statistics of a level monitor can be printed with `print_statistics()`. E.g: `waitingline.length.print_statistics()`:

```
Statistics of Length of waitingline at     50000
                        all    excl.zero         zero
-------------- ------------ ------------ ------------
duration           50000        48499.381     1500.619
mean                   8.427        8.687
std.deviation          4.852        4.691

minimum                0            1
median                 9           10
90% percentile        14           14
95% percentile        16           16
maximum               21           21
```

And, a histogram can be printed with `print_histogram()`. E.g.

```
waitingline.length.print_histogram(30, 0, 1)
```

```
Histogram of Length of waitingline
                        all    excl.zero         zero
-------------- ------------ ------------ ------------
duration           50000        48499.381     1500.619
mean                   8.427        8.687
std.deviation          4.852        4.691

minimum                0            1
median                 9           10
90% percentile        14           14
95% percentile        16           16
```

```
maximum              21          21

           <=      duration     %  cum%
        0          1500.619   3.0   3.0 **|
        1          2111.284   4.2   7.2 ***  |
        2          3528.851   7.1  14.3 *****      |
        3          4319.406   8.6  22.9 ******            |
        4          3354.732   6.7  29.6 *****                  |
        5          2445.603   4.9  34.5 ***                        |
        6          2090.759   4.2  38.7 ***                           |
        7          2046.126   4.1  42.8 ***                               |
        8          1486.956   3.0  45.8 **                                  |
        9          2328.863   4.7  50.4 ***                                     |
       10          4337.502   8.7  59.1 ******                                         |
       11          4546.145   9.1  68.2 *******                                               |
       12          4484.405   9.0  77.2 *******                                                      |
       13          4134.094   8.3  85.4 ******                                                              |
       14          2813.860   5.6  91.1 ****                                                                    |
       15          1714.894   3.4  94.5 **                                                                         |
       16           992.690   2.0  96.5 *                                                                            |
       17           541.546   1.1  97.6                                                                               |
       18           625.048   1.3  98.8 *                                                                              |
       19           502.291   1.0  99.8                                                                                |
       20            86.168   0.2 100.0                                                                                |
       21             8.162   0.0 100                                                                                   |
       22             0       0   100                                                                                   |
       23             0       0   100                                                                                   |
       24             0       0   100                                                                                   |
       25             0       0   100                                                                                   |
       26             0       0   100                                                                                   |
       27             0       0   100                                                                                   |
       28             0       0   100                                                                                   |
       29             0       0   100                                                                                   |
       30             0       0   100                                                                                   |
          inf         0       0   100
```

If neither number_of_bins, nor lowerbound nor bin_width are specified, the histogram will be autoscaled.

Histograms can be printed with their values, instead of bins. This is particularly useful for non numeric tallied values, like names of production stages. For example

```
Histogram of Status
duration           300

value                      duration     %
idle                         70       23.3 ******************
package                      42       14.0 ***********
prepare                      48       16   ************
stage A                      12        4   ***
stage B                      50       16.7 *************
stage C                      54       18   **************
stage D                      24        8   ******
```

8.3 Merging of monitors

Monitors can be merged, to create a new monitor, nearly always to collect aggregated data.

The method Monitor.merge() is used for that, like

```
mc = m0.merge(m1, m2)
```

Then we can just get the mean of the monitors m0, m1 and m2 combined by

```
mc.mean()
```

,but also directly with :

    m0.merge(m1, m2).mean()

Alternatively, monitors can be merged with the + operator, like

```
mc = m0 + m1 + m2
```

And then get the mean of the aggregated monitors with

```
mc.mean()
```

, but also with

    (m0 + m1 + m2).mean()

It is also possible to use the sum function to merge a number of monitors. So

```
print(sum((m0, m1, m2)).mean())
```

Finally, if ms = (m0, m1, m2), it is also possible to use

```
print(sum(ms).mean())
```

A practical example of this is the case where the list waitinglines contains a number of queues.

Then to get the aggregated statistics of the length of all these queues, use

```
sum(waitingline.length for waitingline in waitinglines).print_statistics()
```

For non level monitors, all of the tallied x-values are copied from the to be merged monitors. For level monitors, the x-values are summed, for all the periods where all the monitors were on. Periods where one or more monitors were disabled, are excluded. Note that the merge only takes place at creation of the (timestamped) monitor and not dynamically later.

Sample usage:

Suppose we have three types of products (a, b, c) and that each have a queue for processing, so a.processing, b.processing, c.processing. If we want to print the histogram of the combined (=summed) length of these queues

```
a.processing.length.merge(b.processing.length, c.processing.length, name='combined processing length')).print_histogram()
```

and to get the minimum of the length_of_stay for all queues

```
(a.processing.length_of_stay + b.processing.length_of_stay + c.processing.length_of_stay).minimum()
```

Note that it is possible to rename a merged monitor (particularly those created with + or sum) with the rename() method:

```
sum(waitingline.length for waitingline in waitinglines).rename('aggregated length of waitinglines').print_statistics()
```

Merged monitors are disabled and cannot be enabled again.

8.4 Slicing of monitors

It is possible to slice a monitor with Monitor.slice(), which has two applications:

- to get statistics on a monitor with respect to a given time period, most likely a subrun

- to get statistics on a monitor with respect to a recurring time period, like hour 0-1, hour 0-2, etc.

Examples

```
for i in range(10):
    start = i * 1000
    stop = (i+1) * 1000
    print(f'mean length of q in [{start},{stop})={q.length.slice(start,stop).mean()}'
    print(f'mean length of stay in [{start},{stop})={q.length_of_stay.slice(start,stop).mean()}'

for i in range(24):
    print(f'mean length of q in hour {i}={q.length.slice(i, i+1, 24).mean()}'
    print(f'mean length of stay of q in hour {i}={q.length_of_stay.slice(i, i+1, 24).mean()}'
```

Instead of slice(), a monitor can be sliced as well with the standard slice operator [], like

```
q.length[1000:2000].print_histogram()
q.length[2:3:24].print_histogram()
print(q.length[1000].mean())
```

Note that it is possible to rename a sliced monitor (particularly those created []) with the rename() method:

```
waitingline.length[1000:2000].rename('length of waitingline between t=1000 and t-2000').print_statistics()
```

Sliced monitors are disabled and cannot be enabled again.

8.5 Using monitored values in other packages, like matplotlib

For high quality, reproduction ready, graphs, it can be useful to use additional packages, most notably matplotlib.

The sampled values from a non level monitor can be retrieved with Monitor.x(). If the moment of the sample is required as well, either Monitor.xt() or Monitor.tx() can be used.

For level monitors, there is choice of :

- Monitor.xt()
- Monitor.tx()
- Monitor.xduration()

To get a proper display of a level monitor, we advise something like

```
plt.plot(*waitingline.length.tx(), drawstyle="steps-post")
```

8.6 Pickling a monitor

Monitor.freeze() returns a ‘frozen’ monitor that can be used to store the results not depending on the current environment.

This is particularly useful for pickling a monitor.

E.g. use

```
with open("mon.pickle", "wb") as f:
    pickle.dump(f, mon.freeze())
```

to save the monitor mon, and

```
with open("mon.pickle", "rb") as f:
    mon_retrieved = pickle.load(f)
```

to retrieve the monitor, later.

Both level and non level monitors are supported. Frozen monitors get the name of the original monitor padded with ‘.frozen’ unless specified differently.

8.7 Stats_only monitors

Monitors can optionally collect statistics only.

This minimizes memory usage as individual tallies will not be stored in that case. But beware that some important functionality is not available in stats_only monitors (see below).

You can define the stats_only status at creation time of the monitor, like

```
m = sim.Monitor('m', stats_only=True)
```

But, it is also possible to reset a monitor with a different stats_only value. This can be useful if you want a system monitor, like Component.mode or Component.status to not keep individual tallied values

```
class Car(sim.Component):
    def setup(self):
        self.mode.reset(stats_only=True)
```

When stats_only is active, values are always forced to numeric, with a fallback value of 0.

Monitor with stats_only=True support __call__/get (without arguments), base_name, deregister, duration, duration_zero, maximum, mean, minimum, monitor, name, number_of_entries, number_of_entries_zero, print_histogram, print_histograms, print_statistics, register, rename, reset, reset_monitors, sequence_number, setup, stats_only, std, t, tally, weight, weight_zero

Monitors with stats_only=True do NOT support (these will raise a NotImplementedError) __call__/get (with an argument), arithmeric operations (+, *, /)``, animate, bin_duration, bin_number_of_entries, bin_weight, freeze, histogram_autoscale, median, merge, multiply, percentile, slice, slicing, to_days, to_hours, to_microseconds, to_milliseconds, to_minutes, to_seconds, to_time_unit, to_weeks, to_years, tx, value_duration, value_number_of_entries, value_weight, values, x, xduration, xt, xweight

In line with the above, Queue.reset_monitors, Resource.reset_monitors, State.reset_monitors have a stats_only parameter, with which all monitors can set/reset the stats_only status. So, if you want all eight monitors belonging to a resource, but the requesters' to be stats_only, you can write

```
r.reset_monitors(stats_only=True)
r.requesters.monitors(stats_only=False)
```

The current stats_only mode can be queried with Monitor.stats_only() .

CHAPTER

NINE

DISTRIBUTIONS

9.1 Introduction

Salabim can be used with the standard random module, but it is usually easier to use the salabim distributions.

Internally, salabim uses the random module. There is always a seed associated with each distribution, which is normally random.random.

When a new environment is created, the random seed 1234567 will be set by default. However, it is possible to override this behaviour with the random_seed parameter:

- any hashable value, to set another seed
- null string (“”): no reseeding
- “*”: true random, non reproducible (based on current time)
- None: equivalent to 1234567

It is possible to (re)set the random seed also with the `Environment.random_seed()` method.

As a distribution is an instance of a class, it can be used in assignment, parameters, etc. E.g.

```
inter_arrival_time = sim.Uniform(10,15)
```

And then, to wait for a time sampled from this distribution

```
yield self.hold(inter_arrival_time.sample())
```

or

```
yield self.hold(inter_arrival_time())
```

or

```
yield self.hold(sim.Uniform(10,15).sample())
```

or

```
yield self.hold(sim.Uniform(10,15)())
```

Furthermore, all time related parameters of component methods accept the distribibution itself instead of a float (from a sample). That means that in the above setup, we can also say

```
yield self.hold(inter_arrival_time)
```

or

```
yield self.hold(sim.Uniform(10,15))
```

All distributions are a subclass of _Distribution which supports the following methods:

- mean()
- sample()
- direct calling as an alternative to sample, like Uniform(12,15)()
- bounded_sample() # see below

9.2 Expressions with distributions

It is possible to build up a distribution with an expression containing one or more distributions. Examples

```
d0 = 5 - sim.Uniform(1, 2)  # equivalent to Uniform (3, 4)
d1 = sim.Normal(4, 1) // 1  # integer samples of a normal distribution
arrival_dis = sim.Pdf((0, 1, 2, 3, 4, 5, 6), (18, 18, 18, 18, 18, 8,2), 'days') + sim.Cdf((0,0, 8,10, 17, 90, 24, 100), 'hours')
  # this generates an arrival moment during the week, with emphasis on day 0-4. The distribution over the day concentrates␣
↪between hour 8 and 17.
```

These will make an instance of the class _Expresssion, which can be used as any other distribution

```
arrival_dis.sample()
(sim.IntUniform(1,5) * 10).sample()  # this will return 10, 20, 30, 40 or 50.
(1 / sim.Uniform(1, 2))()  # this will return values between 0.5 and 1 (not uniform!)
```

Like all distributions, the _Expression class supports the mean(), sample(), bounded_sample() and print_info() methods. If the mean can't be calculated, nan will be returned

```
(sim.Uniform(1, 2) / 10).mean()  # 0.15
(sim.Uniform(1, 2) + sim.Uniform(1, 2)).mean()  # 3
(10 / sim.Uniform(1, 2)).mean()  # nan
(sim.Uniform(1, 2) / sim.Uniform(1, 2)).mean()  # nan
```

Note that the expression may contain only the operator +, -, , /, *// and* *. Functions are not allowed. However the `int` function can be emulated with floor division (\\), as is in

```
d1 = sim.Normal(4, 1) // 1  # integer samples of a normal distribution
```

9.3 Bounded sampling

The class Bounded can be used to force a sampled value from a distribution to be within given bounds.

This is realized by checking if the sampled value is within these bounds. If not, another value is sampled, until the sample meets the requirements. If after, 100 retries (customizable) the sampled value does still not meet the requirements, a fail_value will be returned.

Examples

```
dis = sim.Bounded(sim.Normal(3, 1), lowerbound=0)
sample = dis.sample()  # normal distribution, non negative
sim.Bounded(sim.Exponential(6, upperbound=20).sample()  # exponential distribution <= 20
sim.Bounded(sim.Exponential(6, upperbound=20)()  # exponential distribution <= 20
```

It is alo possible to use the bounded_sample() method, with similar functionality. However, the Bounded class is prefered.

9.4 Use of time units in a distribution specification

All distributions apart from Poisson and Beta have an additional parameter, time_unit. If the time_unit is specified at initialization of Environment(), the time_unit of the distribution can now be specified.

As an example, suppose env has been initialized with `env = sim.Environment(time_unit='hours')`. If we then define a duration distribution as

```
duration_dis = sim.Uniform(10, 20, 'days')
```

, the distribution is effectively uniform between 240 and 480 (hours).

This facility makes specification of duration distribution easy and intuitive.

9.5 Available distributions

9.5.1 Beta

Beta distribution with a given

- alpha (shape)
- beta (shape)

E.g.

```
processing_time = sim.Beta(2,4)  # Beta with alpha=2, beta=4`
```

9.5.2 Constant

No sampling is required for this distribution, as it always returns the same value. E.g.

```
processing_time = sim.Constant(10)
```

9.5.3 Erlang

Erlang distribution with a given

- shape (k)
- rate (lambda) or scale (mu)

E.g.

```
inter_arrival_time = sim.Erlang(2, rate=2)  # Erlang-2, with lambda = 2
```

9.5.4 Exponential

Exponential distribution with a given

- mean or rate (lambda)

E.g.

```
inter_arrival_time = sim.Exponential(10)  # on an average every 10 time units
```

9.5.5 Gamma

Gamma distribution with given

- shape (k)
- scale (teta) or rate (beta)

E.g.

```
processing_time = sim.Gamma(2,3)  # Gamma with k=2, teta=3
```

9.5.6 IntUniform

Integer uniform distribution between a given

- lowerbound (inclusive)
- upperbound (inclusive)

E.g.

```
die = sim.IntUniform(1, 6)
```

9.5.7 Normal

Normal distribution with a given

- mean
- standard deviation

E.g.

```
processing_time = sim.Normal(10, 2)  # Normal with mean=10, standard deviation=2
```

Note that this might result in negative values, which might not correct if it is a duration. In that case, use the Bound class to force a non negative value, like

```
yield self.hold(sim.Bounded(sim.Normal(10, 2), 0)())
yield self.hold(sim.Bounded(sim.Normal(10, 2), 0))
```

Normally, sampling is done with the random.normalvariate method. Alternatively, the random.gauss method can be used, by specifying use_gauss=True.

9.5.8 Poisson

Poisson distribution with a given lambda

E.g.

```
occurences_in_one_hour = sim.Poisson(10)  # Poisson distribution with lambda (and thus mean) = 10
```

9.5.9 Triangular

Triangular distribution with a given

- lowerbound
- upperbound
- mode

E.g.

```
processing_time = sim.Triangular(5, 15, 8)
```

9.5.10 Uniform

Uniform distribution between a given

- lowerbound
- upperbound

E.g.

```
processing_time = sim.Uniform(5, 15)
```

9.5.11 Weibull

Weibull distribution with given

- scale (alpha or k)
- shape (beta or lambda)

E.g.

```
time_between_failure = sim.Weibull(2, 5)  # Weibull with k=2. lambda=5
```

9.5.12 Cdf

Cumulative distribution function, specified as a list or tuple with x[i],p[i] values, where p[i] is the cumulative probability that xn<=pn. E.g.

```
processingtime = sim.Cdf((5, 0, 10, 50, 15, 90, 30, 95, 60, 100))
```

This means that 0% is <5, 50% is < 10, 90% is < 15, 95% is < 30 and 100% is <60.

Note: It is required that p[0] is 0 and that p[i]<=p[i+1] and that x[i]<=x[i+1].

It is not required that the last p[] is 100, as all p[]'s are automatically scaled. This means that the two distributions below are identical to the first example

```
processingtime = sim.Cdf((5, 0.00, 10, 0.50, 15, 0.90, 30, 0.95, 60, 1.00))
processingtime = sim.Cdf((5, 0, 10, 10, 15, 18, 30, 19, 60, 20))
```

9.5.13 Pdf

Probability density function, specified as:

1. list or tuple of x[i], p[i] where p[i] is the probability (density)
2. list or tuple of x[i] followed by a list or tuple p[i]
3. list or tuple of x[i] followed by a scalar (value not important)

Note: It is required that the sum of p[i]'s is **greater than** 0.

E.g.

```
processingtime = sim.Pdf((5, 10, 10, 50, 15, 40))
```

This means that 10% is 5, 50% is 10 and 40% is 15.

It is not required that the sum of the p[i]'s is 100, as all p[]'s are automatically scaled. This means that the two distributions below are identical to the first example

```
processingtime = sim.Pdf((5, 0.10, 10, 0.50, 15, 0.40))
processingtime = sim.Pdf((5, 2, 10, 10, 15, 8))
```

And the same with the second form

```
processingtime = sim.Pdf((5, 10, 15), (10, 50, 40))
```

If all x[i]'s have the same probability, the third form is very useful

```
dice = sim.Pdf((1,2,3,4,5,6),1)  # the distribution IntUniform(1,6) does the job as well
dice = sim.Pdf(range(1,7), 1)  # same as above
```

x[i] may be of any type, so it possible to use

```
color = sim.Pdf(('Green', 45, 'Yellow', 10, 'Red', 45))
cartype = sim.Pdf(ordertypes,1)
```

If the x-value is a salabim distribution, not the distribution but a sample of that distribution is returned when sampling

```
processingtime = sim.Pdf((sim.Uniform(5, 10), 50, sim.Uniform(10, 15), 40, sim.Uniform(15, 20), 10))
proctime=processingtime.sample()
```

Here proctime will have a probability of 50% being between 5 and 10, 40% between 10 and 15 and 10% between 15 and 20.

Pdf supports also sampling a number of items from a pdf without replacement. In that case, the probabilities for all items have to be the same. If that is the case, multiple sampling can be done by specifying the number of items to sampled as a parameters to sample.

Examples

```
colors_dis = sim.Pdf(("red", "green", "blue", "yellow"), 1)
colors_dis.sample(4)  # e.g. ["yellow", "green", "blue", "red"]
colors_dis.sample(2)  # e.g. ["green", "blue"]
colors_dis,sample(1)  # e.g. ["blue"], so not "blue" !
```

9.5.14 CumPdf

Probability density function, specified as:

1. list or tuple of x[i], p[i] where p[i] is the cumulative probability (density)
2. list or tuple of x[i] followed by a list or tuple of probabilities p[i]

Note: It is required that p[i]<=p[i+1].

E.g.

```
processingtime = sim.CumPdf((5, 10, 10, 60, 15, 100))
urgent =  sim.CumPdf(True, 0.9, False, 1.0)
```

This means that 10% is 5, 50% is 10 and 40% is 15.

It is not required that the sum of the p[i]'s is 100, as all p[]'s are automatically scaled. This means that the two distributions below are identical to the first example

```
processingtime = sim.CumPdf((5, 0.10, 10, 0.60, 15, 1.00))
processingtime = sim.CumPdf((5, 2, 10, 12, 15, 20))
```

And the same with the second form

```
processingtime = sim.CumPdf((5, 10, 15), (10, 60, 100))
```

x[i] may be of any type, so it possible to use

```
color = sim.CumPdf(('Green', 45, 'Red', 100))
```

If the x-value is a salabim distribution, not the distribution but a sample of that distribution is returned when sampling

```
processingtime = sim.CumPdf((sim.Uniform(5, 10), 50, sim.Uniform(10, 15), 90, sim.Uniform(15, 20), 100))
proctime=processingtime.sample()
```

Here proctime will have a probability of 50% being between 5 and 10, 40% between 10 and 15 and 10% between 15 and 20.

9.5.15 External

The External distribution makes it possible to use a distribution from the modules

- random
- numpy.random
- scipy.stats

as were it a salabim distribution.

The class takes at least one parameter (dis) that specifies the distribution to use, like

- random.uniform
- numpy.random.uniform
- scipy.stats.uniform

Next, all postional and keyword parameters are used for sampling. In case of random and numpy.random by calling the method itself, in case of a scipy.stats distribution by calling rvs().

Examples :: import random d = sim.External(random.lognormvariate, mu=5, sigma=1)

import numpy d = sim.External(numpy.random.laplace, loc=5, scale=1)

import scipy.stats as st d = sim.External(st.stats.beta, a=1, b=2 loc=4, scale=1)

Then sampling with

```
d.sample()
```

or

```
d()
```

If the size is given (only for numpy.random and scipy.stats distributions), salabim will return the sampled values successively. This can be useful to increase performance.

The mean() method returns the proper mean for scipy.stats distributions, nan otherwise.

If a time_unit parameter is given, the sampled values will be multiplied by the applicable factor, e.g.

```
env = sim.Environment(time_unit='seconds')
d = sim.External(st.norm, loc=2, time_unit='minutes')
print(d.mean())
```

will print out

```
120
```

9.5.16 Distribution

A special distribution is the Distribution class. Here, a string will contain the specification of the distribution. This is particularly useful when the distributions are specified in an external file. E.g.

```
with open('experiment.txt', 'r') as f:
    interarrivaltime = sim.Distribution(read(f))
    processingtime = sim.Distribution(read(f))
    numberofparcels = sim.Distribution(read(f))
    delay = sim.Distribution(read(f))
```

With a file experiment.txt

```
Uniform(10, 15)
Triangular(1, 5, 2, time_unit='minutes')
IntUniform(10, 20)
External(random.uniform, 2, 6)
```

or with abbreviation

```
Uni(10,15)
Tri(1, 5, 2, time_unit='minutes')
Int(10, 20)
Extern(random.uniform, 2, 6)
```

or even

```
U(10, 15)
T(1, 5, 2, time_unit='minutes')
I(10, 20)
Ext(random.uniform, 2, 6)
```

The specification of sim.Distribution also accepts a time_unit parameter. Note that if the specification string contains a time_unit parameter as well, the time_unit parameter of Distribution is ignored, e.g.

```
d = sim.Distribution('uniform(1, 2)', time_unit='minutes'))  # 1-2 minutes
d = sim.Distribution('uniform(1, 2, time_unit='hours')'))  # 1-2 hours, same as before
d = sim.Distribution('uniform(1, 2, time_unit='hours')', time_unit='minutes'))  # 1-2 hours, ignore minutes
```

9.5.17 Map

With the Map distribution is it possible to map the sampled values of a distribution to a given function. E.g.

```
round_normal = sim.Map(sim.Normal(10, 4), lambda x: round(x))
```

or, equivalently

```
round_normal = sim.Map(sim.Normal(10, 4), round)
```

The sampled values from the normal distribution will be rounded in either case.

Another example

```
positive_normal = sim.Map(sim.Normal(10, 4), lambda x: x if x > 0 else 0)
```

In this case, negative sampled values will be set to zero, positive values are not affected.

Note that this is different from

```
sim.Bounded(sim.Normal(10, 4), lowerbound=0)
```

as in the latter case resampling will be done when a negative value is sampled, in other words, effectively no zero samples.

CHAPTER

TEN

MISCELLANEOUS

10.1 Run control

Normally, a simulation is run for a given duration with

```
env.run(duration=100)
```

If you do not specify a till or duration parameter, like :: env.run()

, the simulation will run till there are no events left, or otherwise infinitely.

If it required that the simulation does not stop when there are no more events, which can be useful for animation, issue

```
env.run(till=sim.inf)
```

Finally, it is possible to return control to 'main' from a component with

```
env.main().activate()
```

For instance, if we want to stop a simulation after 50 ships are created

```
class ShipGenerator(sim.Component):
    def process(self):
        for _ in range(50):
            yield self.hold(iat)
            Ship()
        env.main().activate
```

Or, if you want to terminate a run based upon a condition

```
class RunChecker(sim.Component):
    def process(self):
        while True:
            if len(q0) + len(q1) > 10:
                env.main.activate()
            yield self.standby()
```

It is perfectly possible and sometimes very useful to continue a simulation after a run statement, like

```
env.run(100)
q.reset_statistics()
env.run(1000)
q.print_statistics()
```

The salabim time (now) can be reset to 0 (or another time) with

```
env.reset_now()
```

Please note that in this case, user time values has to be corrected accordingly.

10.2 Time units

By default, salabim time does not have a specific dimension, which means that is up to the modeller what time unit is used, be it seconds, hours, days or whatever.

It can be useful to work in specific time unit, as this opens the possibility to specify times and durations in another time unit.

In order to use time unit, the environment has to be initialized with a time_unit parameter, like

```
env = sim.Environment(time_unit='hours')
```

From then on, the simulation runs in hours. Standard output is in then in hours and for instance

```
self.enter(q)
yield self.hold(48)
print(env.now() - self.queuetime())
```

means hold for 48 (hours) and 48 will be printed.

But, now we also specify a time in another time unit and get times in a specific time unit

```
self.enter(q)
yield self.hold(env.days(2))
print(env.to_minutes(env.now() - self.queuetime()))
```

means hold for 2 days = 48 hours and 2880 (48 * 60) will be printed.

With this, it is possible to set the speed of the animation. For instance if we want one second of real time to correspond to 5 minutes

```
env.speed(sim.minutes(5))
```

The following time units are available:

- 'years'
- 'weeks'
- 'days'
- 'hours'
- 'minutes'
- 'seconds'
- 'milliseconds'
- 'microseconds'
- 'n/a' which means nothing is assigned and conversions are not supported

For conversion from a given time unit to the simulation time unit, the following calls are available:

- years()
- weeks()
- days()
- hours()
- minutes()
- seconds()
- milliseconds()
- microseconds()

For conversion from the simulation time unit to a given time unit, the following calls are available:

- to_years()
- to_weeks()
- to_days()
- to_hours()
- to_minutes()
- to_seconds()
- to_milliseconds()
- to_microseconds()
- to_time_unit()

Distributions (apart from IntUniform, Poisson and Beta) can also specify the time unit, like

```
env = sim.Environment(time_unit='seconds')
processingtime_dis = sim.Uniform(10, 20, 'minutes')
dryingtime_dis = sim.Normal(2, 0.1, 'hours')
```

Note that distribution keep the time unit information and therefore salabim is able to present information on the distribution in pretty format: So

```
processingtime_dis.print_info()
dryingtime_dis.print_info()
```

will print

```
Uniform distribution 0x2bfb4d14c50
  lowerbound=10 minutes
  upperbound=20 minutes
  randomstream=0x2bfb2cc0c78
Normal distribution 0x2bfb42ce630
  mean=2 hours
  standard_deviation=0.1 hours
  coefficient_of_variation=0.05
  randomstream=0x2bfb2cc0c78
```

It is possible to scale a non level monitor (particularly length_of_stay of a queue) to another time unit, which is particularly useful for print_histogram. For example

```
waitingline.length_of_stay.to_minutes().print_histogram()
```

will use minutes as the time unit.

This is equivalent to

```
waitingline.length_of_stay.to_time_unit('minutes').print_histogram()
```

And if the environment's time unit is seconds, equivalent to

```
(waitingline.length_of_stay * 60).print_histogram()
```

10.3 Working with datetimes and timedeltas

Salabim allways uses a float based time scale.

In some instances it might be useful to deal with datetimes or timedelta, e.g. when getting input from an external source (file, API, . . . _ or when writing out times in a more human readable format.

In order to work with datetimes, datetime0 should be specified at creation of an environment or later with a call to Environment.datetime0.

```
env = sim.Environment(datetime0=True)
```

If we now have a file in the form

```
2022-04-30 12:00:04 100 red
2022-04-30 12:05:02 200 blue
2022-04-30 13:56:12 200 red
```

we could use the following code to read and process this information

```
class WorkLoadGenerator(sim.Component):
    def process(self):
        with open("workload.txt", "r") as f:
            for line in f.readlines():
                workload_date_str = line[:19]
                workload_datetime = datetime.datetime.strptime(workload_date_str, "%Y-%m-%d %H:%M:%S")
                yield self.hold(till=env.datetime_to_t(workload_datetime))
                # generate workload
```

```
env = sim.Environment(datetime0=True, trace=True)
WorkLoadGenerator()
env.run()
```

This generates the following output

```
line#                       time current component    action                               information
------ ----------------------- --------------------- ------------------------------------ ----------------------------------------
↪---------
                                                      line numbers refers to               salabim_exp.py
   43                                                 default environment initialize
   43                                                 main create
   43  Thu 1970-01-01 00:00:00 main                   current
   44                                                 workloadgenerator.0 create
   44                                                 workloadgenerator.0 activate         scheduled for Thu 1970-01-01 00:00:00 @␣
↪  35  process=process
   45                                                 main run                             ends on no events left  @   45+
   35  Thu 1970-01-01 00:00:00 workloadgenerator.0    current
   40                                                 workloadgenerator.0 hold +19112 12:00:04 scheduled for Sat 2022-04-30␣
↪12:00:04 @   40+
   40+ Sat 2022-04-30 12:00:04 workloadgenerator.0    current
   40                                                 workloadgenerator.0 hold +00:04:58   scheduled for Sat 2022-04-30 12:05:02 @␣
↪  40+
   40+ Sat 2022-04-30 12:05:02 workloadgenerator.0    current
   40                                                 workloadgenerator.0 hold +01:51:10   scheduled for Sat 2022-04-30 13:56:12 @␣
↪  40+
   40+ Sat 2022-04-30 13:56:12 workloadgenerator.0    current
   40+                                                workloadgenerator.0 ended
   45+                                                run ended                            no events left
   45+ Sat 2022-04-30 13:56:12 main                   current
```

As you can see, the simulation starts on 1 January 1970.

This can be changed by specifying the start datettime

```
env = sim.Environment(datetime0=datetime.datetime(2022, 4, 29), trace=True)
```

Once a datetime0 is specified, the moments and durations in the trace will be like shown above. Note that the time in the upper right hand corner of an animation is also presented like this.

If no time unit is specified and datetime0 is given, salabim will assume that the time unit is seconds. But, it is also possible to work in any time unit combined with datetime0

```
env = sim.Environment(time_unit="days", datetime0=datetime.datetime(2022, 4, 29))
env.trace(True)
env.run(1)  # this is one day
```

gives

```
49                                          main run +1 00:00:00                scheduled for Sat 2022-04-30 00:00:00 @ ␣
↪49+
49+ Sat 2022-04-30 00:00:00 main            current
```

There are a couple of additional methods connected to datetime0:

- *datetime_to_t*
- *timedelta_to_duration*
- *Environment.'t_to_datetime*
- *Environment.'duration_to_timedelta*
- *Environment.'datetime0*

Refer to the reference sections for details.

10.4 Usage of the the trace facility

10.4.1 Control

Tracing can be turned on at time of creating an environment

```
env = sim.Environment(trace=True)
```

and can be turned on during a simulation run with `env.trace(True)` and likewise turned off with `env.trace(False)`. The current status can be queried with `env.trace(False)`.

It is also possible to direct the trace output to file, by using

```
with open('output.txt', 'w') as out:
    env = sim.Environment(trace=out)
```

or

```
out = open('output.txt', 'w')
env.trace(out)
...
out.close()
```

10.4.2 Interpretation of the trace

A trace ouput looks like

```
line#        time current component    action                               information
-----  ---------- -------------------- -----------------------------------  ------------------------------------------------
                                       line numbers refers to               Example - basic.py
   11                                  default environment initialize
   11                                  main create
   11       0.000 main                 current
   12                                  car.0 create
   12                                  car.0 activate                       scheduled for      0.000 @    6  process=process
   13                                  main run                             scheduled for      5.000 @   13+
    6       0.000 car.0                current
    8                                  car.0 hold                           scheduled for      1.000 @    8+
    8+      1.000 car.0                current
    8                                  car.0 hold                           scheduled for      2.000 @    8+
    8+      2.000 car.0                current
    8                                  car.0 hold                           scheduled for      3.000 @    8+
    8+      3.000 car.0                current
    8                                  car.0 hold                           scheduled for      4.000 @    8+
    8+      4.000 car.0                current
    8                                  car.0 hold                           scheduled for      5.000 @    8+
   13+      5.000 main                 current
```

The texts are pretty self explanatory. If a mode is given, that will be shown as well.

Note that line numbers sometimes has an added +, which means that the activation is actually the statement following the given line number. When there is more than one source file involved, the line number may be preceded by a letter. In that case, the trace will contain an information line to which file that letter refers to.

The text 'process=' refers to the activation process, which is quite often just process.

When a time is followed by an exclamation mark (!), it means that the component is scheduled urgent, i.e. before all other events for the same moment.

10.4.3 Suppressing components from being shown in the trace

It is possible to suppress the trace when a specific component becomes or is current. This can be either indicated at creation of a component with

```
c = sim.Component(suppress_trace=True)
```

or later with

```
c.suppress_trace(True)
```

Note that this suppresses all trace output during the time a component is current.

10.4.4 Showing standby components in the trace

By default standby components are (apart from when they become non standby) suppressed from the trace. With `env.suppress_trace_standby(False)` standby components are fully traced.

10.4.5 Changing the format of times and durations

By default, times and durations are printed as 10.3f.

But it is possible to use other formats by overriding the *Environment.time_to_str* and *Environment,duration_to_str* methods.

E.g. :: sim.Environment.time_to_str = lambda self, t: f"{t:10.1f}

or ::

> **class MyEnvironment(sim.Environment):**
>
> > **def time_to_str(self, t):** return datetime.datetime.utcfromtimestamp(ts).strftime("%Y-%m-%d %H:%M:%S")
>
> …
>
> env = sim.MyEnvironment(trace=True)

In general, it is recommended to use datetime0 fumctionality in this case (see the section "Working with datetimes and timedeltas").

Note that the method *time_to_str* should return a string with the same length, regardless of the value of t.

10.4.6 Adding lines to the trace output

A model can add additional information to the trace with the `Environment.print_trace()` method. This methods accepts up to five parameters to be show on one line. When trace is False, nothing will be displayed.

Example

```
env.print_trace('', '**ALERT**', 'Houston, we have a problem with', c.name())
```

Refer to the reference section for details.

10.4.7 Redirecting trace output

Normally, all trace output is written to stdout.

However, if the trace parameter of Environment() or Environment.trace() is a file handle (open for write), then the trace output will go to the specified file. Note that it is required to close the file handle at the end, so a context manager (‘with’) is recommended. Example

```
with open('output.txt', 'w') as out:
    env = sim.Environment(trace=out)
    ...
    env.run()
```

Alternatively, stdout can be set to a file, in which case ALL salabim output will be redirected. This can be done like

```
save_stdout = sys.stdout
sys.stdout = open('output.txt', 'w')
```

If required, it is possible to revert to the original stdout

```
sys.stdout.close()
sys.stdout = save_stdout
```

10.4.8 Suppressing line numbers in the trace

Particularly when the trace output is written to a file, it may be useful to suppress line numbers in the trace. This can result in dramatic (up to 50 times!) performance increase, because the calculation of line numbers is very demanding.

The method *Environment.suppress_trace_linenumbers()* can be used to disable/enable line number in the trace

```
env = sim.Environment(trace=True)
env.suppress_trace_linenumbers(True)
```

By default, line numbers are not suppressed in the trace. The current status can be queried with

```
print(env.suppress_trace_linenumbers())
```

CHAPTER

ELEVEN

ANIMATION

Animation is a powerful tool to debug, test and demonstrate simulations.

It is possible to show a number of shapes (lines, rectangles, circles, etc), texts as well (images) in a window. These objects can be dynamically updated. Monitors may be animated by showing the current value against the time. Furthermore the components in a queue may be shown in a highly customizable way. As text animation may be dynamically updated, it is even possible to show the current state, (monitor) statistics, etc. in the animation windows.

Salabim's animation engine also allows some user input.

It is important to realize that animation calls can be still given when animation is actually off. In that case, there is hardly any impact on the performance.

Salabim animations can be

- synchronized with the simulation clock and run in real time (synchronized)
- advanced per simulation event (non synchronized)

In synchronized mode, one time unit in the simulation can correspond to any period in real time, e.g.

- 1 time unit in simulation time –> 1 second real time (speed = 1) (default)
- 1 time unit in simulation time –> 4 seconds real time (speed = 0.25)
- 4 time units in simulation time –> 1 second real time (speed = 4)

The most common way to start an animation is by calling ‘‘ env.animate(True)‘‘ or with a call to `animation_parameters(animate=True)`.

Animations can be started and stopped during execution (i.e. run). When main is active, the animation is always stopped.

The animation uses a coordinate system that -by default- is in screen pixels. The lower left corner is (0,0). But, the user can change both the coordinate of the lower left corner (translation) as well as set the x-coordinate of the lower right hand corner (scaling). Note that x- and y-scaling are always the same. Furthermore, it is possible to specify the colour of the background with `animation_parameters`.

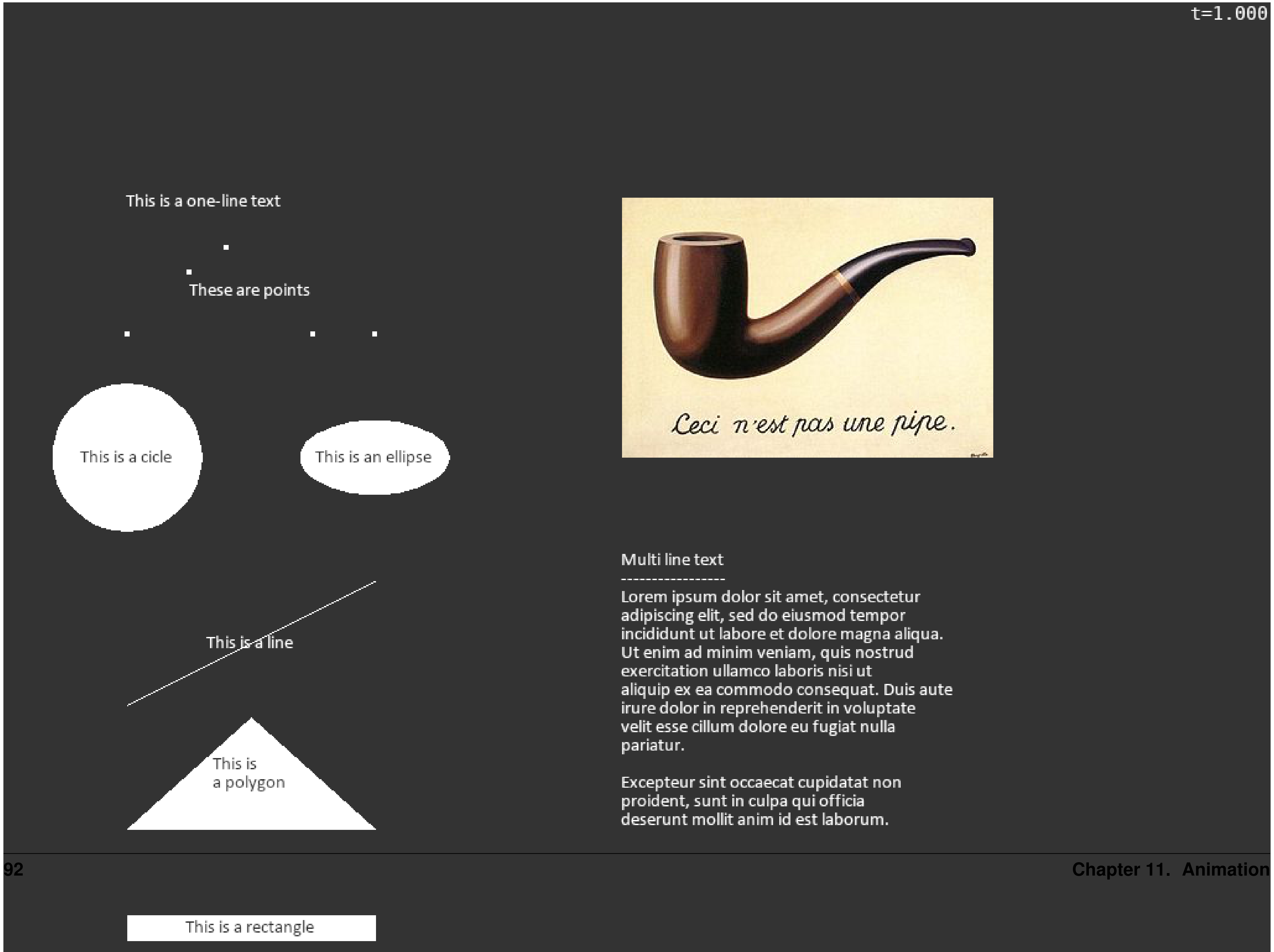
Prior to version 2.3.0 there was actually just one animation object class: Animate. This interface is described later as the new animation classes are easier to use and even offer some additional functionality.

New style animation classes can be used to put texts, rectangles, polygon, lines, series of points, circles or images on the screen. All types can be connected to an optional text.

Here is a sample program to show of all the new style animation classes:

```
# Animate classes.py

"""
This program demonstrates the various animation classes available in salabim.
"""
import salabim as sim

env = sim.Environment(trace=False)
env.animate(True)
env.modelname("Demo animation classes")
env.background_color("20%gray")

sim.AnimatePolygon(spec=(100, 100, 300, 100, 200, 190), text="This is\na polygon")
sim.AnimateLine(spec=(100, 200, 300, 300), text="This is a line")
sim.AnimateRectangle(spec=(100, 10, 300, 30), text="This is a rectangle")
sim.AnimateCircle(radius=60, x=100, y=400, text="This is a cicle")
sim.AnimateCircle(radius=60, radius1=30, x=300, y=400, text="This is an ellipse")
sim.AnimatePoints(spec=(100, 500, 150, 550, 180, 570, 250, 500, 300, 500), text="These are points")
sim.AnimateText(text="This is a one-line text", x=100, y=600)
sim.AnimateText(
    text="""\
Multi line text
-----------------
Lorem ipsum dolor sit amet, consectetur
adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud
exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat. Duis aute
irure dolor in reprehenderit in voluptate
velit esse cillum dolore eu fugiat nulla
pariatur.

Excepteur sint occaecat cupidatat non
proident, sunt in culpa qui officia
deserunt mollit anim id est laborum.
""",
    x=500,
    y=100,
)

sim.AnimateImage("Pas un pipe.jpg", x=500, y=400)
env.run(100)
```

Resulting in:



Multi line text

Lorem ipsum dolor sit amet, consectetur
adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud
exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat. Duis aute
irure dolor in reprehenderit in voluptate
velit esse cillum dolore eu fugiat nulla
pariatur.

Excepteur sint occaecat cupidatat non
proident, sunt in culpa qui officia
deserunt mollit anim id est laborum.

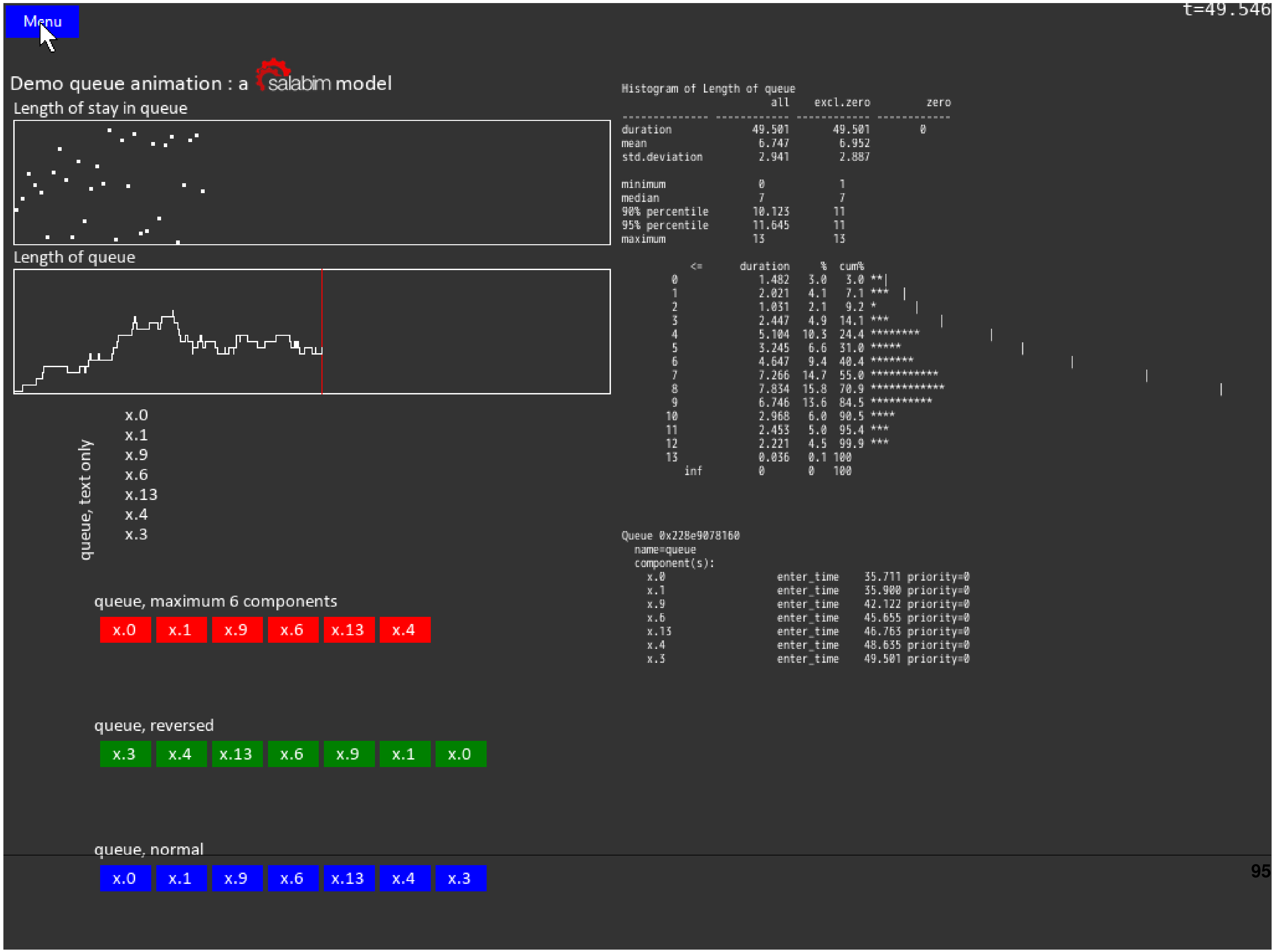
Animation of the components of a queue is accomplished with `AnimateQueue()`. It is possible to use the standard shape of components, which is a rectangle with the sequence number or define your own shape(s). The queue can be build up in west, east, north or south directions. It is possible to limit the number of component shown.

Monitors can be visualized dynamically with `AnimateMonitor()`.

These features are demonstrated in *Demo queue animation.py*

```
import salabim as sim

'''
This is a demonstration of several ways to show queues dynamically and the corresponding statistics
The model simply generates components that enter a queue and leave after a certain time.

Note that the actual model code (in the process description of X does not contain any reference
to the animation!
'''


class X(sim.Component):
    def setup(self, i):
        self.i = i

    def animation_objects(self, id):
        '''
        the way the component is determined by the id, specified in AnimateQueue
        'text' means just the name
        any other value represents the colour
        '''
        if id == 'text':
            ao0 = sim.AnimateText(text=self.name(), textcolor='fg', text_anchor='nw')
            return 0, 16, ao0
        else:
            ao0 = sim.AnimateRectangle((-20, 0, 20, 20),
                text=self.name(), fillcolor=id, textcolor='white', arg=self)
            return 45, 0, ao0

    def process(self):
        while True:
            yield self.hold(sim.Uniform(0, 20)())
            self.enter(q)
            yield self.hold(sim.Uniform(0, 20)())
            self.leave()


env = sim.Environment(trace=False)
env.background_color('20%gray')

q = sim.Queue('queue')

qa0 = sim.AnimateQueue(q, x=100, y=50, title='queue, normal', direction='e', id='blue')
qa1 = sim.AnimateQueue(q, x=100, y=250, title='queue, maximum 6 components', direction='e', max_length=6, id='red')
qa2 = sim.AnimateQueue(q, x=100, y=150, title='queue, reversed', direction='e', reverse=True, id='green')
qa3 = sim.AnimateQueue(q, x=100, y=440, title='queue, text only', direction='s', id='text')

sim.AnimateMonitor(q.length, x=10, y=450, width=480, height=100, horizontal_scale=5, vertical_scale=5)

sim.AnimateMonitor(q.length_of_stay, x=10, y=570, width=480, height=100, horizontal_scale=5, vertical_scale=5)

sim.AnimateText(text=lambda: q.length.print_histogram(as_str=True), x=500, y=700,
    text_anchor='nw', font='narrow', fontsize=10)

sim.AnimateText(text=lambda: q.print_info(as_str=True), x=500, y=340,
    text_anchor='nw', font='narrow', fontsize=10)

[X(i=i) for i in range(15)]
env.animate(True)
env.modelname('Demo queue animation')
env.run()
```

Here is snapshot of this powerful, dynamics (including the histogram!):



```
Histogram of Length of queue
                        all    excl.zero         zero
-------------- ----------- ----------- -----------
duration             49.501       49.501        0
mean                  6.747        6.952
std.deviation         2.941        2.887

minimum               0            1
median                7            7
90% percentile       10.123       11
95% percentile       11.645       11
maximum              13           13

           <=      duration     %  cum%
            0         1.482   3.0   3.0 **|
            1         2.021   4.1   7.1 ***  |
            2         1.031   2.1   9.2 *      |
            3         2.447   4.9  14.1 ***        |
            4         5.104  10.3  24.4 ********           |
            5         3.245   6.6  31.0 *****                  |
            6         4.647   9.4  40.4 *******                        |
            7         7.266  14.7  55.0 ***********                                |
            8         7.834  15.8  70.9 ************                                           |
            9         6.746  13.6  84.5 **********
           10         2.968   6.0  90.5 ****
           11         2.453   5.0  95.4 ***
           12         2.221   4.5  99.9 ***
           13         0.036   0.1 100
          inf         0       0   100
```

```
Queue 0x228e9078160
  name=queue
  component(s):
    x.0                enter_time    35.711 priority=0
    x.1                enter_time    35.900 priority=0
    x.9                enter_time    42.122 priority=0
    x.6                enter_time    45.655 priority=0
    x.13               enter_time    46.763 priority=0
    x.4                enter_time    48.635 priority=0
    x.3                enter_time    49.501 priority=0
```

11.1 Advanced

The various classes have a lot of parameters, like color, line width, font, etc.

These parameters can be given just as a scalar, like:

```
sim.AnimateText(text='Hello world', x=200, y=300, textcolor='red')
```

But each of these parameters may also be a:

- function with zero arguments
- function with one argument being the time t
- function with two arguments being 'arg' and the time t
- a method with instance 'arg' and the time t

The function or method is called at each animation frame update (maximum of 30 frames per second).

This makes it for instance possible to show dynamically the mean of monitor m, like in

```
sim.AnimateRectangle(spec=(10, 10, 200, 30), text=lambda:  str(m.mean())
```

11.2 Class Animate

This class can be used to show:

- line (if line0 is specified)
- rectangle (if rectangle0 is specified)
- polygon (if polygon0 is specified)
- circle (if circle0 is specified)
- text (if text is specified)
- image (if image is specified)

Note that only one type is allowed per instance of Animate.

Nearly all attributes of an Animate object are interpolated between time t0 and t1. If t0 is not specified, now() is assumed. If t1 is not specified inf is assumed, which means that the attribute will be the '0' attribute.

E.g.:

`Animate(x0=100,y0=100,rectangle0==(-10,-10,10,10))` will show a square around (100,100) for ever `Animate(x0=100,y0=100,x1=200,y1=0, rectangle0=(-10,-10,10,10))` will still show the same square around (100,100) as t1 is not specified `Animate(t1=env.now()+10,x0=100,y0=100, x1=200,y1=0,rectangle0=(-10,-10,10,10))` will show a square moving from (100,100) to (200,0) in 10 units of time.

It also possible to let the rectangle change shape over time:

`Animate(t1=env.now(),x0=100,y0=100,x1=200,y1=0,rectangle0=(-10,-10,10,10),rectangle1=(-20,-20,20,20))` will show a moving and growing rectangle.

By default, the animation object will not change anymore after t1, but will remain visible. Alternatively, if keep=False is specified, the object will disappear at time t1.

Also, colors, fontsizes, angles can be changed in a linear way over time.

E.g.:

`Animate(t1=env.now()+10,text='Test',textcolor0='red',textcolor1='blue',angle0=0,angle1=360)` will show a rotating text changing from red to blue in 10 units of time.

The animation object can be updated with the update method. Here, once again, all the attributes can be specified to change over time. Note that the defaults for the '0' values are the actual values at t=now().

Thus,

`an=Animate(t0=0,t1=10,x0=0,x1=100,y0=0,circle0=(10,),circle1=(20,))` will show a horizontally moving, growing circle.

Now, at time t=5, we issue `an.update(t1=10,y1=50,circle1=(10,))` Then x0 will be set 50 (halfway 0 an 100) and cicle0 to (15,) (halfway 10 and 20). Thus the circle will shrink to its original size and move vertically from (50,0) to (50,50). This concept is very useful for moving objects whose position and orientation are controlled by the simulation.

Here we explain how an attribute changes during time. We use x as an example. Normally, x=x0 at t=t0 and x=x1 at t>=t1. between t=t0 and t=t1, x is linearly interpolated. An application can however override the x method. The prefered way is to subclass the Animate class:

```python
# Demo animate 1
import salabim as sim


class AnimateMovingText(sim.Animate):
    def __init__(self):
        sim.Animate.__init__(self, text="", x0=100, x1=1000, y0=100, t1=env.now() + 10)

    def x(self, t):
        return sim.interpolate(sim.interpolate(t, self.t0, self.t1, 0, 1) ** 2, 0, 1, self.x0, self.x1)

    def y(self, t):
        return int(t) * 50

    def text(self, t):
        return "{:0.1f}".format(t)


sim.reset()
env = sim.Environment()

env.animate(True)

AnimateMovingText()

env.run(till=sim.inf)  # otherwise the simulation will end at t=0, because there are no events left
```

This code will show the current simulation time moving from left to right, uniformly accelerated. And the text will be shown a bit higher up, every second. It is not necessary to use t0, t1, x0, x1, but is a convenient way of setting attributes.

The following methods may be overridden:

method	circle	image	line	polygon	rectangle	text
anchor		•				
angle	•	•	•	•	•	•
circle	•					
fillcolor	•			•	•	
fontsize						•
image		•				
layer	•	•	•	•	•	•
line			•			
linecolor	•		•	•	•	
linewidth	•		•	•	•	
max_lines						•
offsetx	•	•	•	•	•	•
offsety	•	•	•	•	•	•
polygon				•		
rectangle					•	
text						•
text_anchor						•
textcolor						•

Dashboard animation

Here we present an example model where the simulation code is completely separated from the animation code. This makes communication and debugging and switching off animation much easier.

The example below generates 15 persons starting at time 0, 1, ... These persons enter a queue called q and stay there 15 time units.

The animation dashboard shows the first 10 persons in the queue q, along with the length of that q.

```
# Demo animate 2.py
import salabim as sim


class AnimateWaitSquare(sim.Animate):
    def __init__(self, i):
        self.i = i
        sim.Animate.__init__(
            self, rectangle0=(-12, -10, 12, 10), x0=300 - 30 * i, y0=100, fillcolor0="red", linewidth0=0
        )

    def visible(self, t):
        return q[self.i] is not None


class AnimateWaitText(sim.Animate):
    def __init__(self, i):
        self.i = i
        sim.Animate.__init__(self, text="", x0=300 - 30 * i, y0=100, textcolor0="white")

    def text(self, t):
        component_i = q[self.i]

        if component_i is None:
            return ""
        else:
            return component_i.name()


def do_animation():
    env.animate(True)
    for i in range(10):
        AnimateWaitSquare(i)
        AnimateWaitText(i)
    show_length = sim.Animate(text="", x0=330, y0=100, textcolor0="black", anchor="w")
    show_length.text = lambda t: "Length= " + str(len(q))


class Person(sim.Component):
    def process(self):
        self.enter(q)
        yield self.hold(15)
        self.leave(q)


env = sim.Environment(trace=True)

q = sim.Queue("q")
for i in range(15):
    Person(name="{:02d}".format(i), at=i)

do_animation()

env.run()
```

All animation initialization is in `do_animation`, where first 10 rectangle and text Animate objects are created. These are classes that are inherited from sim.Animate.

The AnimateWaitSquare defines a red rectangle at a specific position in the `sim.Animate.__init__()` call. Note that normally these squares should be displayed. But, here we have overridden the visible method. If there is no i-th component in the q, the square will be made invisible. Otherwise, it is visible.

The AnimateWaitText is more or less defined in a similar way. It defines a text in white at a specific position. Only the text method is overridden and will return the name of the i-th component in the queue, if any. Otherwise the null string will be returned.

The length of the queue q could be defined also by subclassing sim.Animate, but here we just make a direct instance of Animate with the null string as the text to be displayed. And then we immediately override the text method with a lambda function. Note that in this case, self is not available!

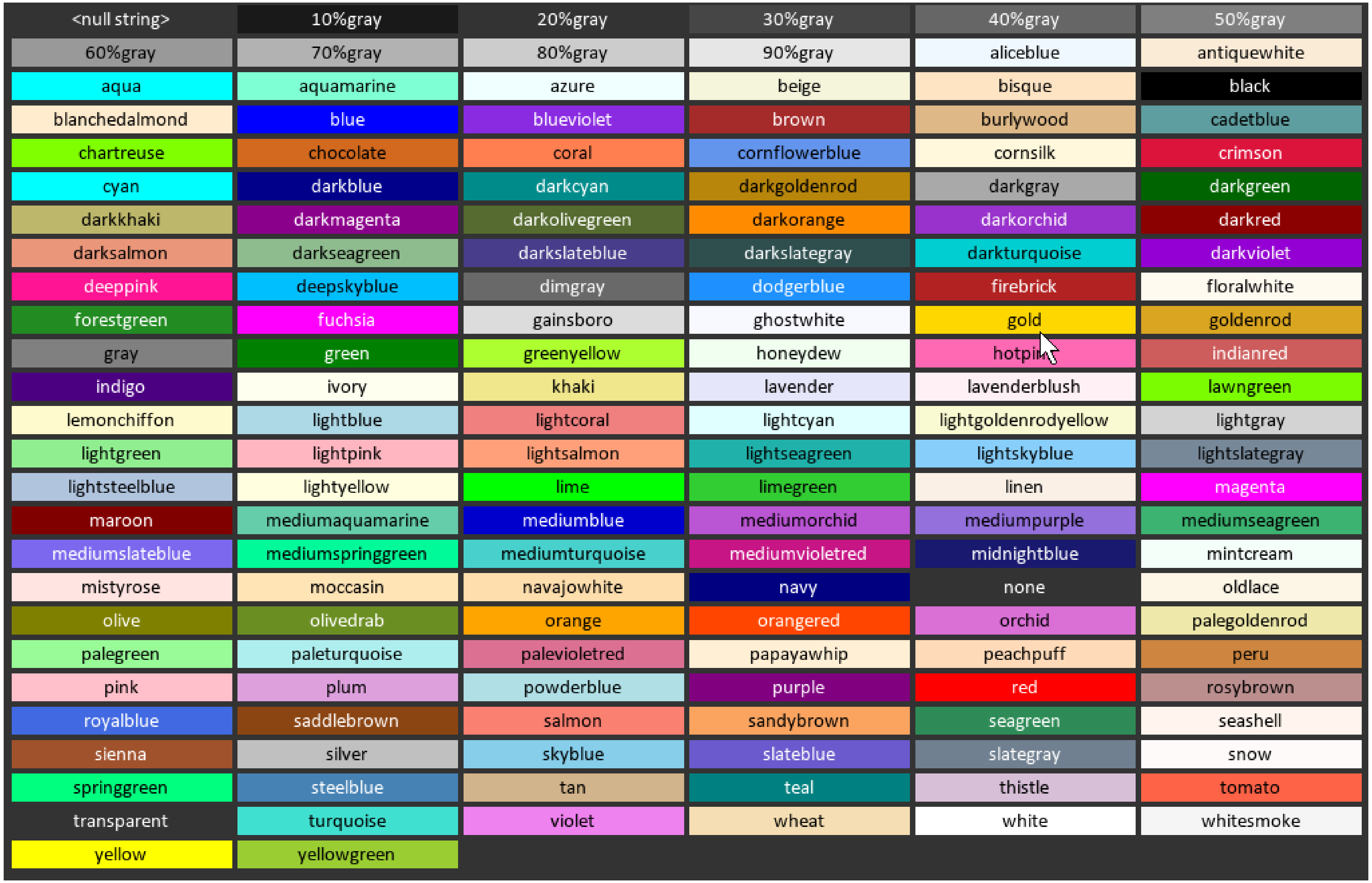
11.3 Using colours

When a colour has to be specified in one of the animation methods, salabim offers a choice of specification:

- *#rrggbb* rr, gg, bb in hex, alpha=255
- *#rrggbbaa* rr, gg, bb, aa in hex, alpha=aa

- *(r, g, b)* r, g, b in 0-255, alpha=255
- *(r, g, b, a)* r, g, b in 0-255, alpha=a
- *“fg”* current foreground color
- *“bg”* current background color
- *colorname* alpha=255
- *colorname, a* alpha=a

The colornames are defined as follows:



<null string>	10%gray	20%gray	30%gray	40%gray	50%gray
60%gray	70%gray	80%gray	90%gray	aliceblue	antiquewhite
aqua	aquamarine	azure	beige	bisque	black
blanchedalmond	blue	blueviolet	brown	burlywood	cadetblue
chartreuse	chocolate	coral	cornflowerblue	cornsilk	crimson
cyan	darkblue	darkcyan	darkgoldenrod	darkgray	darkgreen
darkkhaki	darkmagenta	darkolivegreen	darkorange	darkorchid	darkred
darksalmon	darkseagreen	darkslateblue	darkslategray	darkturquoise	darkviolet
deeppink	deepskyblue	dimgray	dodgerblue	firebrick	floralwhite
forestgreen	fuchsia	gainsboro	ghostwhite	gold	goldenrod
gray	green	greenyellow	honeydew	hotpi	indianred
indigo	ivory	khaki	lavender	lavenderblush	lawngreen
lemonchiffon	lightblue	lightcoral	lightcyan	lightgoldenrodyellow	lightgray
lightgreen	lightpink	lightsalmon	lightseagreen	lightskyblue	lightslategray
lightsteelblue	lightyellow	lime	limegreen	linen	magenta
maroon	mediumaquamarine	mediumblue	mediumorchid	mediumpurple	mediumseagreen
mediumslateblue	mediumspringgreen	mediumturquoise	mediumvioletred	midnightblue	mintcream
mistyrose	moccasin	navajowhite	navy	none	oldlace
olive	olivedrab	orange	orangered	orchid	palegoldenrod
palegreen	paleturquoise	palevioletred	papayawhip	peachpuff	peru
pink	plum	powderblue	purple	red	rosybrown
royalblue	saddlebrown	salmon	sandybrown	seagreen	seashell
sienna	silver	skyblue	slateblue	slategray	snow
springgreen	steelblue	tan	teal	thistle	tomato
transparent	turquoise	violet	wheat	white	whitesmoke
yellow	yellowgreen				

This output can be generated with the following program:

```
# Show colornames

import salabim as sim

env = sim.Environment()
names = sorted(sim.colornames().keys())
env.modelname("show colornames")
env.background_color("20%gray")
env.animate(True)
x = 10
y = env.height() - 110
sx = 165
sy = 21

for name in names:
    sim.Animate(rectangle0=(x, y, x + sx, y + sy), fillcolor0=name)
    sim.Animate(
        text=(name, "<null string>")[name == ""],
        x0=x + sx / 2,
        y0=y + sy / 2,
        anchor="c",
        textcolor0=("black", "white")[env.is_dark(name)],
        fontsize0=15,
    )
    x += sx + 4
    if x + sx > 1024:
        y -= sy + 4
        x = 10

env.run(sim.inf)
```

11.4 Running animations on PyDroid3

In order to run animations on PyDroid3 platforms, it required that the main program imports tkinter

```
import tkinter
```

Note that it can't harm to include this import on non PyDroid3 platforms, apart from Pythonista, where tkinter is not available. In order to make a platform independent animation, you could use

```
if not sim.Pythonista:
    import tkinter
```

or

```
try:
    import tkinter
except ImportError:
    pass
```

11.5 Avoiding crashes in tkinter

When animating a large number of objects, it is possible that tkinter crashes because there are too many tkinter bitmaps aka canvas objects, sometimes by issuing a 'Fail to allocate bitmap', sometimes without any message. Salabim limits the number of bitmap automatically by combining animation objects in one aggregated bitmap if the number of bitmaps exceeds a given maximum. Unfortunately it is not possible to detect this 'Fail to allocate bitmap', so it may take some experimentation to find a workable maximum (maybe going as low as 1000).

By default, salabim sets the maximum number of bitmaps to 4000, but may be changed with the Environment.maximum_number_of_bitmaps() method, or the maximum_number_of_bitmaps parameter of Environment.animation_parameters(). Choosing a too low maximum (particularly 0), may result in a performance degradation. The bitmap aggregation process is transparent to the user.

Note that does this not apply to the Pythonista implementation, where bitmaps are always aggregated.

11.6 Video production and snapshots

An animation can be recorded as an .mp4 (not under PyDroid3) or .avi video by specifying `video=filename` in the call to animation_parameters, or issue `video(filename)`. The effect is that 30 times per second (scaled animation time) a frame is written. In this case, the animation does not run synchronized with the wall clock any more. Depending on the complexity of the animation, the simulation might run faster of slower than real time. In contrast to an ordinary animation, frames are never skipped.

The video *has* to be closed explicity with

```
env.video(False)
```

or it is possible to use a context manager to automatically close a video file, like:

```
with env.video('myvideo.mp4'):
    ...
    env.run(10)
```

This will automatically close the file myvideo.mp4 upon leaving the with block.

It is also possible to create an animated gif or animated png files by specifying a .gif or .png file. In that case, repeat and pingpong are additional options. Note that animated gif/png are considerable bigger than ordinary video files. So, try and limit the length to 10 seconds. Animated pngs may be written with a transparent background (alpha < 255).

Video production supports also the creation of a series of individual frames, in .jpg, .gif, .png, .tiff or .bmp format. In this case, the video name has to contain an asterisk (*) which will be expanded at runtime to a 6 digit zero padded frame number, e.g.

```
env.video('test*.jpg')
```

will write individual autonumbered frames named

```
test000000.jpg
test000001.jpg
test000002.jpg
...
```

Prior to creating the frames, all files matching the specification will be removed, in order to get only the required frames, most likely for post processing with ffmpeg or similar.

Note that individual frame video production and animated gif/png production are available on all platforms, including Pythonista.

Salabim also supports taking a snapshot of an animated screen with `Environment.snapshot()`.

11.7 Video creation on machines that do not support tkinter

On some servers, tkinter is not available. In that case it is still possible to create videos. That can be done by setting the `blind_animation=True` in the call to `sim.Environment`.

Note that this can also be used to (slightly) increase the performance of video production.

11.8 Audio support

On Windows platforms, it is possible to add an audio track to a video. With `Environment.audio()` an audio track (usually an mp3 file) will be added. The audio may stopped by issueing `audio("")`. If another audio is started, the current audio, if any, will be stopped.

Adding audio to a video requires that ffmpeg is installed and in the search path. Refer to *www.ffmpeg.org* for downloads and instructions.

In order to develop lip synced videos, it is possible to play audios parallel to a simulation, provided the animation speed is equal to the audio_speed (1 by default). Audio playback is supported on Pythonista and Windows platforms only.

CHAPTER

TWELVE

3D ANIMATION

3D animation provides a way to visualize simulations in 3D.

Salabim supports the 'classical' 2D animation along with a 3D animation. The 3D animation is particularly useful for showing the functionality of a system to domain experts, where objects are moving up and down, like stacking systems, multi storey facilities and complex road infrastructure. But also just to make a visualy more attractive presentation (video).

3D animations always run in parallel with the tkinter (2D) window, to control the simulation (pausing, exiting, etc). Also custom buttons may be installed in that window. And of course, a 2D animation as well.

As with 2D animations, animation calls can be still given when animation is actually off. In that case, there is hardly any impact on the performance.

The 3D animation uses its own coordinate system.

All 3D animation objects are so called new style animation classes.

The following 3D classes are available as of now:

- Animate3dBox
- Animate3dBar
- Animate3dCylinder
- Animate3dRectangle
- Animate3dLine
- Animate3dObj

On top of that, animation of the components of a queue in 3D is accomplished with `Animate3dQueue()`. It is possible to use the standard shape of components, which is a box of size 1 in all directions. The queue can be build up in +x, -x, +y, -y, +z, -z direction. It is possible to limit the number of component shown.

It is possible to overlay a 2d animation object (like AnimateText) over the 3D windows. This is done by adding (over3d=True) to the Animatexxx call. This feature is particularly useful for videos.

CHAPTER

THIRTEEN

READING ITEMS FROM A FILE

Salabim models often need to read input values from a file.

As these data are quite often quite unstructured, using the standard read facilities of text files can be rather tedious.

Therefore, salabim offers the possibility to read a file item by item.

Example usage

```
with sim.ItemFile(filename) as f:
    run_length = f.read_item_float()
    run_name = f.read_item()
```

Or (not recommended)

```
f = sim.InputFile(filename)
run_length = f.read_item_float()
run_name = f.read_item()
f.close()
```

The input file is read per item, where blanks, linefeeds, tabs are treated as separators. Any text on a line after a # character is ignored. Any text within curly brackets ({}) is ignored (and treated as an item separator). Note that this strictly on a per line basis. If a blank or tab is to be included in a string, use single or double quotes. The recommended way to end a list of values is //

So, a typical input file is

```
# Typical experiment file for a salabim model
1000              # run length
'Experiment 2.0'  # run name
```

```
#Model           speed color
#--------------- ----- ------

'Peugeot 208'        150 red
'Peugeot 3008'       175 orange
'Citroen C5'         160 blue
'Renault "Twingo"'   165 green
//

France {country} Europe {continent}

#end of file
```

Instead of the filename as a parameter to ItemFile, also a string with the content can be given. In that case, at least one linefeed has to be in the content string. Usually, the content string will be triple quoted. This can be very useful during testing as the input is part of the source file and not external, e.g.

```
test_input = '''
one two
three four
five
'''
with sim.ItemFile(test_input) as f:
    while True:
        try:
            print(f.read_item())
        except EOFError:
            break
```

CHAPTER

FOURTEEN

REFERENCE

14.1 3D Animation

class salabim.**Animate3dBase**(*visible=True*, *keep=True*, *arg=None*, *layer=0*, *parent=None*, *env=None*, ***kwargs*)

Base class for a 3D animation object When a class inherits from this base class, it will be added to the animation objects list to be shown

Parameters

- **visible** (*bool*) – visible if False, animation object is not shown, shown otherwise (default True)
- **layer** (*int*) – layer value lower layer values are displayed later in the frame (default 0)
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)
- **parent** (Component) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **env** (Environment) – environment where the component is defined if omitted, default_env will be used

remove()

removes the 3d animation oject

setup()

called immediately after initialization of a the Animate3dBase object.

by default this is a dummy method, but it can be overridden.

only keyword arguments will be passed

Example

class AnimateVehicle(sim.Animate3dBase):

def setup(self, length): self.length = length self.register_dynamic_attributes("length")

. . .

show (*unremove*)
It is possible to use this method if already shown

class salabim.**Animate3dBox** (*x_len=1, y_len=1, z_len=1, x=0, y=0, z=0, z_angle=0, x_ref=0, y_ref=0, z_ref=0, color='white', edge_color='', shaded=False, visible=True, arg=None, layer=0, parent=None, env=None, **kwargs*)
Creates a 3D box

Parameters

- **x_len** (*float*) – length of the box in x direction (deffult 1)
- **y_len** (*float*) – length of the box in y direction (default 1)
- **z_len** (*float*) – length of the box in z direction (default 1)
- **x** (*float*) – x position of the box (default 0)
- **y** (*float*) – y position of the box (default 0)
- **z** (*float*) – z position of the box (default 0)
- **z_angle** (*float*) – angle around the z-axis (default 0)
- **x_ref** (*int*) – if -1, the x parameter refers to the 'end' of the box if 0, the x parameter refers to the center of the box (default) if 1, the x parameter refers to the 'start' of the box
- **y_ref** (*int*) – if -1, the y parameter refers to the 'end' of the box if 0, the y parameter refers to the center of the box (default) if 1, the y parameter refers to the 'start' of the box
- **z_ref** (*int*) – if -1, the z parameter refers to the 'end' of the box if 0, the z parameter refers to the center of the box (default) if 1, the z parameter refers to the 'start' of the box
- **color** (*colorspec*) – color of the box (default "white") if the color is "" (or the alpha is 0), the sides will not be colored at all
- **edge_color** (*colorspec*) – color of the edges of the (default "") if the color is "" (or the alpha is 0), the edges will not be drawn at all
- **shaded** (*bool*) – if False (default), all sides will be colored with color if True, the various sides will have a sligtly different darkness, thus resulting in a pseudo shaded object

- **visible** (*bool*) – visible if False, animation object is not shown, shown otherwise (default True)
- **layer** (*int*) – layer value lower layer values are displayed later in the frame (default 0)
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)
- **parent** (Component) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **env** (Environment) – environment where the component is defined if omitted, default_env will be used

Note: All parameters, apart from parent, arg and env can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: my_x - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

class salabim.**Animate3dCylinder**(*x0=0, y0=0, z0=0, x1=1, y1=1, z1=1, color='white', radius=1, number_of_sides=8, rotation_angle=0, show_lids=True, visible=True, arg=None, layer=0, parent=None, env=None, **kwargs*)

Creates a 3D cylinder between two given points

Parameters

- **x0** (*float*) – x coordinate of start point (default 0)
- **y0** (*float*) – y coordinate of start point (default 0)
- **z0** (*float*) – z coordinate of start point (default 0)
- **x1** (*float*) – x coordinate of end point (default 0)
- **y1** (*float*) – y coordinate of end point (default 0)
- **z1** (*float*) – z coordinate of end point (default 0)
- **color** (*colorspec*) – color of the cylinder (default "white")
- **number_of_sides** (*int*) – number of sides of the cylinder (default 8) must be >= 3
- **rotation_angle** (*float*) – rotation of the bar in degrees (default 0)
- **show_lids** (*bool*) – if True (default), the lids will be drawn if False, tyhe cylinder will be open at both sides
- **visible** (*bool*) – visible if False, animation object is not shown, shown otherwise (default True)
- **layer** (*int*) – layer value lower layer values are displayed later in the frame (default 0)

- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)
- **parent** (Component) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **env** (Environment) – environment where the component is defined if omitted, default_env will be used

Note: All parameters, apart from parent, arg and env can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: my_x - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

class salabim.**Animate3dGrid**(*x_range=[0], y_range=[0], z_range=[0], color='white', visible=True, arg=None, layer=0, parent=None, env=None, **kwargs*)

Creates a 3D grid

Parameters

- **x_range** (*iterable*) – x coordinates of grid lines (default [0])
- **y_range** (*iterable*) – y coordinates of grid lines (default [0])
- **z_range** (*iterable*) – z coordinates of grid lines (default [0])
- **color** (*colorspec*) – color of the line (default "white")
- **visible** (*bool*) – visible if False, animation object is not shown, shown otherwise (default True)
- **layer** (*int*) – layer value lower layer values are displayed later in the frame (default 0)
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)
- **parent** (Component) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **env** (Environment) – environment where the component is defined if omitted, default_env will be used

Note: All parameters, apart from parent, arg and env can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: my_x - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

class salabim.**Animate3dLine**(*x0=0*, *y0=0*, *z0=0*, *x1=1*, *y1=1*, *z1=0*, *color='white'*, *visible=True*, *arg=None*, *layer=0*, *parent=None*, *env=None*, ***kwargs*)

Creates a 3D line

Parameters

- **x0** (*float*) – x coordinate of start point (default 0)
- **y0** (*float*) – y coordinate of start point (default 0)
- **z0** (*float*) – z coordinate of start point (default 0)
- **x1** (*float*) – x coordinate of end point (default 0)
- **y1** (*float*) – y coordinate of end point (default 0)
- **z1** (*float*) – z coordinate of end point (default 0)
- **color** (*colorspec*) – color of the line (default "white")
- **visible** (*bool*) – visible if False, animation object is not shown, shown otherwise (default True)
- **layer** (*int*) – layer value lower layer values are displayed later in the frame (default 0)
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)
- **parent** (Component) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **env** (Environment) – environment where the component is defined if omitted, default_env will be used

Note: All parameters, apart from parent, arg and env can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: my_x - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

class salabim.**Animate3dObj**(*filename*, *x=0*, *y=0*, *z=0*, *x_angle=0*, *y_angle=0*, *z_angle=0*, *x_translate=0*, *y_translate=0*, *z_translate=0*, *x_scale=1*, *y_scale=1*, *z_scale=1*, *show_warnings=False*, *visible=True*, *arg=None*, *layer=0*, *parent=None*, *env=None*, ***kwargs*)

Creates a 3D animation object from an .obj file

Parameters

- **filename** (*str or Path*) – obj file to be read (default extension .obj) if there are .mtl or .jpg required by this file, they should be available
- **x** (*float*) – x position (default 0)

- **y** (*float*) – y position (default 0)
- **z** (*float*) – z position (default 0)
- **x_angle** (*float*) – angle along x axis (default: 0)
- **y_angle** (*float*) – angle along y axis (default: 0)
- **z_angle** (*float*) – angle along z axis (default: 0)
- **x_translate** (*float*) – translation in x direction (default: 0)
- **y_translate** (*float*) – translation in y direction (default: 0)
- **z_translate** (*float*) – translation in z direction (default: 0)
- **x_scale** (*float*) – scaling in x direction (default: 1)
- **y_translate** – translation in y direction (default: 1)
- **z_translate** – translation in z direction (default: 1)
- **show_warnings** (*bool*) – as pywavefront does not support all obj commands, reading the file sometimes leads to (many) warning log messages with this flag, they can be turned off (the deafult)
- **visible** (*bool*) – visible if False, animation object is not shown, shown otherwise (default True)
- **layer** (*int*) – layer value lower layer values are displayed later in the frame (default 0)
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)
- **parent** (Component) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **env** (Environment) – environment where the component is defined if omitted, default_env will be used

Note: All parameters, apart from parent, arg and env can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: my_x - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

Note: This method requires the pywavefront and pyglet module to be installed

class salabim.**Animate3dQueue**(*queue*, *x=0*, *y=0*, *z=0*, *direction='x+'*, *max_length=None*, *reverse=False*, *layer=0*, *id=None*, *arg=None*, *parent=None*, *visible=True*, *keep=True*)

Animates the component in a queue.

Parameters

- **queue** (Queue) –
- **x** (*float*) – x-position of the first component in the queue default: 0
- **y** (*float*) – y-position of the first component in the queue default: 0
- **z** (*float*) – z-position of the first component in the queue default: 0
- **direction** (*str*) – if "x+", waiting line runs in positive x direction (default) if "x-", waiting line runs in negative x direction if "y+", waiting line runs in positive y direction if "y-", waiting line runs in negative y direction if "z+", waiting line runs in positive z direction if "z-", waiting line runs in negative z direction
- **reverse** (*bool*) – if False (default), display in normal order. If True, reversed.
- **max_length** (*int*) – maximum number of components to be displayed
- **layer** (*int*) – layer (default 0)
- **id** (*any*) – the animation works by calling the animation_objects method of each component, optionally with id. By default, this is self, but can be overriden, particularly with the queue
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)
- **visible** (*bool*) – if False, nothing will be shown (default True)
- **keep** (*bool*) – if False, animation object will be taken from the animation objects. With show(), the animation can be reshown. (default True)
- **parent** (Component) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible

Note: All parameters, apart from queue, id, arg and parent can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: title - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

queue()

Returns the queue this object refers to. Can be useful in Component.animation3d_objects

Return type queue

show(*unremove*)
It is possible to use this method if already shown

class salabim.**Animate3dRectangle**(*x0=0*, *y0=0*, *x1=1*, *y1=1*, *z=0*, *color='white'*, *visible=True*, *arg=None*, *layer=0*, *parent=None*, *env=None*, ***kwargs*)
Creates a 3D rectangle

Parameters

- **x0** (*float*) – lower left x position (default 0)
- **y0** (*float*) – lower left y position (default 0)
- **x1** (*float*) – upper right x position (default 1)
- **y1** (*float*) – upper right y position (default 1)
- **z** (*float*) – z position of rectangle (default 0)
- **color** (*colorspec*) – color of the rectangle (default "white")
- **visible** (*bool*) – visible if False, animation object is not shown, shown otherwise (default True)
- **layer** (*int*) – layer value lower layer values are displayed later in the frame (default 0)
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)
- **parent** (Component) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **env** (Environment) – environment where the component is defined if omitted, default_env will be used

Note: All parameters, apart from parent, arg and env can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: my_x - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

14.2 Animation

class salabim.**Animate**(*parent=None*, *layer=0*, *keep=True*, *visible=True*, *screen_coordinates=None*, *t0=None*, *x0=0*, *y0=0*, *offsetx0=0*, *offsety0=0*, *circle0=None*, *line0=None*, *polygon0=None*, *rectangle0=None*, *points0=None*, *image=None*, *text=None*, *font=”*, *anchor=’c’*, *as_points=False*, *max_lines=0*, *text_anchor=None*, *linewidth0=None*, *fillcolor0=None*, *linecolor0=’fg’*, *textcolor0=’fg’*, *angle0=0*, *alpha0=255*, *fontsize0=20*, *width0=None*, *t1=None*, *x1=None*, *y1=None*, *offsetx1=None*, *offsety1=None*, *circle1=None*, *line1=None*, *polygon1=None*, *rectangle1=None*, *points1=None*, *linewidth1=None*, *fillcolor1=None*, *linecolor1=None*, *textcolor1=None*, *angle1=None*, *alpha1=None*, *fontsize1=None*, *width1=None*, *xy_anchor=”*, *over3d=None*, *env=None*)

defines an animation object

Parameters

- **parent** (`Component`) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **layer** (*int*) – layer value lower layer values are on top of higher layer values (default 0)
- **keep** (*bool*) – keep if False, animation object is hidden after t1, shown otherwise (default True)
- **visible** (*bool*) – visible if False, animation object is not shown, shown otherwise (default True)
- **screen_coordinates** (*bool*) – use screen_coordinates normally, the scale parameters are use for positioning and scaling objects. if True, screen_coordinates will be used instead.
- **xy_anchor** (*str*) – specifies where x and y (i.e. x0, y0, x1 and y1) are relative to possible values are (default: sw) : `nw n ne w c e sw s se` If null string, the given coordimates are used untranslated
- **t0** (*float*) – time of start of the animation (default: now)
- **x0** (*float*) – x-coordinate of the origin at time t0 (default 0)
- **y0** (*float*) – y-coordinate of the origin at time t0 (default 0)
- **offsetx0** (*float*) – offsets the x-coordinate of the object at time t0 (default 0)
- **offsety0** (*float*) – offsets the y-coordinate of the object at time t0 (default 0)
- **circle0** (*float or tuple/list*) – the circle spec of the circle at time t0 - radius - one item tuple/list containing the radius - five items tuple/list cntaining radius, radius1, arc_angle0, arc_angle1 and draw_arc (see class AnimateCircle for details)
- **line0** (*tuple*) – the line(s) (xa,ya,xb,yb,xc,yc, . . .) at time t0
- **polygon0** (*tuple*) – the polygon (xa,ya,xb,yb,xc,yc, . . .) at time t0 the last point will be auto connected to the start
- **rectangle0** (*tuple*) – the rectangle (xlowerleft,ylowerleft,xupperright,yupperright) at time t0

- **`image`** (*`str, pathlib.Path or PIL image`*) – the image to be displayed This may be either a filename or a PIL image
- **`text`** (*`str, tuple or list`*) – the text to be displayed if text is str, the text may contain linefeeds, which are shown as individual lines
- **`max_lines`** (*`int`*) – the maximum of lines of text to be displayed if positive, it refers to the first max_lines lines if negative, it refers to the first -max_lines lines if zero (default), all lines will be displayed
- **`font`** (*`str or list/tuple`*) – font to be used for texts Either a string or a list/tuple of fontnames. If not found, uses calibri or arial
- **`anchor`** (*`str`*) – anchor position specifies where to put images or texts relative to the anchor point possible values are (default: c): `nw n ne w c e sw s se`
- **`as_points`** (*`bool`*) – if False (default), lines in line, rectangle and polygon are drawn if True, only the end points are shown in line, rectangle and polygon
- **`linewidth0`** (*`float`*) – linewidth of the contour at time t0 (default 0 for polygon, rectangle and circle, 1 for line) if as_point is True, the default size is 3
- **`fillcolor0`** (*`colorspec`*) – color of interior at time t0 (default foreground_color) if as_points is True, fillcolor0 defaults to transparent
- **`linecolor0`** (*`colorspec`*) – color of the contour at time t0 (default foreground_color)
- **`textcolor0`** (*`colorspec`*) – color of the text at time 0 (default foreground_color)
- **`angle0`** (*`float`*) – angle of the polygon at time t0 (in degrees) (default 0)
- **`alpha0`** (*`float`*) – alpha of the image at time t0 (0-255) (default 255)
- **`fontsize0`** (*`float`*) – fontsize of text at time t0 (default 20)
- **`width0`** (*`float`*) – width of the image to be displayed at time t0 if omitted or None, no scaling
- **`t1`** (*`float`*) – time of end of the animation (default inf) if keep=True, the animation will continue (frozen) after t1
- **`x1`** (*`float`*) – x-coordinate of the origin at time t1(default x0)
- **`y1`** (*`float`*) – y-coordinate of the origin at time t1 (default y0)
- **`offsetx1`** (*`float`*) – offsets the x-coordinate of the object at time t1 (default offsetx0)
- **`offsety1`** (*`float`*) – offsets the y-coordinate of the object at time t1 (default offsety0)
- **`circle1`** (*`float or tuple/list`*) – the circle spec of the circle at time t1 (default: circle0) - radius - one item tuple/list containing the radius - five items tuple/list cntaining radius, radius1, arc_angle0, arc_angle1 and draw_arc (see class AnimateCircle for details)
- **`line1`** (*`tuple`*) – the line(s) at time t1 (xa,ya,xb,yb,xc,yc, . . .) (default: line0) should have the same number of elements as line0
- **`polygon1`** (*`tuple`*) – the polygon at time t1 (xa,ya,xb,yb,xc,yc, . . .) (default: polygon0) should have the same number of elements as polygon0

- **rectangle1** (*tuple*) – the rectangle (xlowerleft,ylowerleft,xupperright,yupperright) at time t1 (default: rectangle0)
- **linewidth1** (*float*) – linewidth of the contour at time t1 (default linewidth0)
- **fillcolor1** (*colorspec*) – color of interior at time t1 (default fillcolor0)
- **linecolor1** (*colorspec*) – color of the contour at time t1 (default linecolor0)
- **textcolor1** (*colorspec*) – color of text at time t1 (default textcolor0)
- **angle1** (*float*) – angle of the polygon at time t1 (in degrees) (default angle0)
- **alpha1** (*float*) – alpha of the image at time t1 (0-255) (default alpha0)
- **fontsize1** (*float*) – fontsize of text at time t1 (default: fontsize0)
- **width1** (*float*) – width of the image to be displayed at time t1 (default: width0)
- **over3d** (*bool*) – if True, this object will be rendered to the OpenGL window if False (default), the normal 2D plane will be used.

Note:

one (and only one) of the following parameters is required:

- circle0
- image
- line0
- polygon0
- rectangle0
- text

colors may be specified as a

- valid colorname
- hexname
- tuple (R,G,B) or (R,G,B,A)
- “fg” or “bg”

colornames may contain an additional alpha, like `red#7f` hexnames may be either 3 of 4 bytes long (`#rrggbb` or `#rrggbbaa`) both colornames and hexnames may be given as a tuple with an additional alpha between 0 and 255, e.g. `(255,0,255,128)`, ("red",127)‘‘ or `("#ff00ff",128)` fg is the foreground color bg is the background color

Permitted parameters

parameter	circle	image	line	polygon	rectangle	text
parent	•	•	•	•	•	•
layer	•	•	•	•	•	•
keep	•	•	•	•	•	•
screen_coordinates	•	•	•	•	•	•
xy_anchor	•	•	•	•	•	•
t0,t1	•	•	•	•	•	•
x0,x1	•	•	•	•	•	•
y0,y1	•	•	•	•	•	•
offsetx0,offsetx1	•	•	•	•	•	•
offsety0,offsety1	•	•	•	•	•	•
circle0,circle1	•					
image		•				
line0,line1			•			
polygon0,polygon1				•		
rectangle0,rectangle1						

alpha(*t=None*)
alpha of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **alpha** – default behaviour: linear interpolation between self.alpha0 and self.alpha1

Return type float

anchor(*t=None*)
anchor of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **anchor** – default behaviour: self.anchor0 (anchor given at creation or update)

Return type str

angle(*t=None*)
angle of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **angle** – default behaviour: linear interpolation between self.angle0 and self.angle1

Return type float

as_points(*t=None*)
as_points of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **as_points** – default behaviour: self.as_points (text given at creation or update)

Return type bool

circle(*t=None*)
circle of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **circle** – either - radius - one item tuple/list containing the radius - five items tuple/list cntaining radius, radius1, arc_angle0, arc_angle1 and draw_arc (see class AnimateCircle for details) default behaviour: linear interpolation between self.circle0 and self.circle1

Return type float or tuple/list

fillcolor (*t=None*)
fillcolor of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **fillcolor** – default behaviour: linear interpolation between self.fillcolor0 and self.fillcolor1

Return type colorspec

font (*t=None*)
font of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **font** – default behaviour: self.font0 (font given at creation or update)

Return type str

fontsize (*t=None*)
fontsize of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **fontsize** – default behaviour: linear interpolation between self.fontsize0 and self.fontsize1

Return type float

image (*t=None*)
image of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **image** – use function spec_to_image to load a file default behaviour: self.image0 (image given at creation or update)

Return type PIL.Image.Image

keep (*t*)
keep attribute of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **keep** – default behaviour: self.keep0 or t <= self.t1 (visible given at creation or update)

Return type bool

layer (*t=None*)
layer of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **layer** – default behaviour: self.layer0 (layer given at creation or update)

Return type int or float

line(*t=None*)
line of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **line** – series of x- and y-coordinates (xa,ya,xb,yb,xc,yc, . . .) default behaviour: linear interpolation between self.line0 and self.line1

Return type tuple

linecolor(*t=None*)
linecolor of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **linecolor** – default behaviour: linear interpolation between self.linecolor0 and self.linecolor1

Return type colorspec

linewidth(*t=None*)
linewidth of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **linewidth** – default behaviour: linear interpolation between self.linewidth0 and self.linewidth1

Return type float

max_lines(*t=None*)
maximum number of lines to be displayed of text. May be overridden.

Parameters **t** (*float*) – current time

Returns **max_lines** – default behaviour: self.max_lines0 (max_lines given at creation or update)

Return type int

offsetx(*t=None*)
offsetx of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **offsetx** – default behaviour: linear interpolation between self.offsetx0 and self.offsetx1

Return type float

offsety(*t=None*)
offsety of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **offsety** – default behaviour: linear interpolation between self.offsety0 and self.offsety1

Return type float

points(*t=None*)
points of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **points** – series of x- and y-coordinates (xa,ya,xb,yb,xc,yc, . . .) default behaviour: linear interpolation between self.points0 and self.points1

Return type tuple

polygon(*t=None*)
polygon of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **polygon** – series of x- and y-coordinates describing the polygon (xa,ya,xb,yb,xc,yc, . . .) default behaviour: linear interpolation between self.polygon0 and self.polygon1

Return type tuple

rectangle(*t=None*)
rectangle of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **rectangle** – (xlowerleft,ylowerlef,xupperright,yupperright) default behaviour: linear interpolation between self.rectangle0 and self.rectangle1

Return type tuple

remove()
removes the animation object from the animation queue, so effectively ending this animation.

Note: The animation object might be still updated, if required

text(*t=None*)
text of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **text** – default behaviour: self.text0 (text given at creation or update)

Return type str

text_anchor(*t=None*)
text_anchor of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **text_anchor** – default behaviour: self.text_anchor0 (text_anchor given at creation or update)

Return type str

textcolor(*t=None*)
textcolor of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **textcolor** – default behaviour: linear interpolation between self.textcolor0 and self.textcolor1

Return type colorspec

update(*layer=None, keep=None, visible=None, t0=None, x0=None, y0=None, offsetx0=None, offsety0=None, circle0=None, line0=None, polygon0=None, rectangle0=None, points0=None, image=None, text=None, font=None, anchor=None, xy_anchor0=None, max_lines=None, text_anchor=None, linewidth0=None, fillcolor0=None, linecolor0=None, textcolor0=None, angle0=None, alpha0=None, fontsize0=None, width0=None, xy_anchor1=None, as_points=None, t1=None, x1=None, y1=None, offsetx1=None, offsety1=None, circle1=None, line1=None, polygon1=None, rectangle1=None, points1=None, linewidth1=None, fillcolor1=None, linecolor1=None, textcolor1=None, angle1=None, alpha1=None, fontsize1=None, width1=None*)
updates an animation object

Parameters

- **layer** (*int*) – layer value lower layer values are on top of higher layer values (default see below)
- **keep** (*bool*) – keep if False, animation object is hidden after t1, shown otherwise (default see below)
- **visible** (*bool*) – visible if False, animation object is not shown, shown otherwise (default see below)
- **xy_anchor** (*str*) – specifies where x and y (i.e. x0, y0, x1 and y1) are relative to possible values are: `nw n ne w c e sw s se` If null string, the given coordimates are used untranslated default see below
- **t0** (*float*) – time of start of the animation (default: now)

- **x0** (*float*) – x-coordinate of the origin at time t0 (default see below)
- **y0** (*float*) – y-coordinate of the origin at time t0 (default see below)
- **offsetx0** (*float*) – offsets the x-coordinate of the object at time t0 (default see below)
- **offsety0** (*float*) – offsets the y-coordinate of the object at time t0 (default see below)
- **circle0** (*float or tuple/list*) – the circle spec of the circle at time t0 - radius - one item tuple/list containing the radius - five items tuple/list cntaining radius, radius1, arc_angle0, arc_angle1 and draw_arc (see class AnimateCircle for details)
- **line0** (*tuple*) – the line(s) at time t0 (xa,ya,xb,yb,xc,yc, . . .) (default see below)
- **polygon0** (*tuple*) – the polygon at time t0 (xa,ya,xb,yb,xc,yc, . . .) the last point will be auto connected to the start (default see below)
- **rectangle0** (*tuple*) – the rectangle at time t0 (xlowerleft,ylowerlef,xupperright,yupperright) (default see below)
- **points0** (*tuple*) – the points(s) at time t0 (xa,ya,xb,yb,xc,yc, . . .) (default see below)
- **image** (*str or PIL image*) – the image to be displayed This may be either a filename or a PIL image (default see below)
- **text** (*str*) – the text to be displayed (default see below)
- **font** (*str or list/tuple*) – font to be used for texts Either a string or a list/tuple of fontnames. (default see below) If not found, uses calibri or arial
- **max_lines** (*int*) – the maximum of lines of text to be displayed if positive, it refers to the first max_lines lines if negative, it refers to the first -max_lines lines if zero (default), all lines will be displayed
- **anchor** (*str*) – anchor position specifies where to put images or texts relative to the anchor point (default see below) possible values are (default: c): `nw n ne` `w c e` `sw s se`
- **linewidth0** (*float*) – linewidth of the contour at time t0 (default see below)
- **fillcolor0** (*colorspec*) – color of interior/text at time t0 (default see below)
- **linecolor0** (*colorspec*) – color of the contour at time t0 (default see below)
- **angle0** (*float*) – angle of the polygon at time t0 (in degrees) (default see below)
- **fontsize0** (*float*) – fontsize of text at time t0 (default see below)
- **width0** (*float*) – width of the image to be displayed at time t0 (default see below) if None, the original width of the image will be used
- **t1** (*float*) – time of end of the animation (default: inf) if keep=True, the animation will continue (frozen) after t1
- **x1** (*float*) – x-coordinate of the origin at time t1 (default x0)

- **y1** (*float*) – y-coordinate of the origin at time t1 (default y0)
- **offsetx1** (*float*) – offsets the x-coordinate of the object at time t1 (default offsetx0)
- **offsety1** (*float*) – offsets the y-coordinate of the object at time t1 (default offset0)
- **circle1** (*float or tuple/ist*) – the circle spec of the circle at time t1 - radius - one item tuple/list containing the radius - five items tuple/list cntaining radius, radius1, arc_angle0, arc_angle1 and draw_arc (see class AnimateCircle for details)
- **line1** (*tuple*) – the line(s) at time t1 (xa,ya,xb,yb,xc,yc, . . .) (default: line0) should have the same number of elements as line0
- **polygon1** (*tuple*) – the polygon at time t1 (xa,ya,xb,yb,xc,yc, . . .) (default: polygon0) should have the same number of elements as polygon0
- **rectangle1** (*tuple*) – the rectangle at time t (xlowerleft,ylowerleft,xupperright,yupperright) (default: rectangle0)
- **points1** (*tuple*) – the points(s) at time t1 (xa,ya,xb,yb,xc,yc, . . .) (default: points0) should have the same number of elements as points1
- **linewidth1** (*float*) – linewidth of the contour at time t1 (default linewidth0)
- **fillcolor1** (*colorspec*) – color of interior/text at time t1 (default fillcolor0)
- **linecolor1** (*colorspec*) – color of the contour at time t1 (default linecolor0)
- **angle1** (*float*) – angle of the polygon at time t1 (in degrees) (default angle0)
- **fontsize1** (*float*) – fontsize of text at time t1 (default: fontsize0)
- **width1** (*float*) – width of the image to be displayed at time t1 (default: width0)

Note: The type of the animation cannot be changed with this method. The default value of most of the parameters is the current value (at time now)

visible(*t=None*)
visible attribute of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **visible** – default behaviour: self.visible0 and t >= self.t0 (visible given at creation or update)

Return type bool

width(*t=None*)
width position of an animated image object. May be overridden.

Parameters **t** (*float*) – current time

Returns **width** – default behaviour: linear interpolation between self.width0 and self.width1 if None, the original width of the image will be used

Return type float

x (*t=None*)
x-position of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **x** – default behaviour: linear interpolation between self.x0 and self.x1

Return type float

xy_anchor (*t=None*)
xy_anchor attribute of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **xy_anchor** – default behaviour: self.xy_anchor0 (xy_anchor given at creation or update)

Return type str

y (*t=None*)
y-position of an animate object. May be overridden.

Parameters **t** (*float*) – current time

Returns **y** – default behaviour: linear interpolation between self.y0 and self.y1

Return type float

class salabim.**AnimateButton** (*x=0*, *y=0*, *width=80*, *fillcolor='fg'*, *color='bg'*, *text="*, *font="*, *fontsize=15*, *action=None*, *env=None*, *xy_anchor='sw'*)
defines a button

Parameters

- **x** (*int*) – x-coordinate of centre of the button in screen coordinates (default 0)
- **y** (*int*) – y-coordinate of centre of the button in screen coordinates (default 0)
- **width** (*int*) – width of button in screen coordinates (default 80)
- **height** (*int*) – height of button in screen coordinates (default 30)
- **linewidth** (*int*) – width of contour in screen coordinates (default 0=no contour)
- **fillcolor** (*colorspec*) – color of the interior (foreground_color)
- **linecolor** (*colorspec*) – color of contour (default foreground_color)

- **color** (*colorspec*) – color of the text (default background_color)
- **text** (*str or function*) – text of the button (default null string) if text is an argumentless function, this will be called each time; the button is shown/updated
- **font** (*str*) – font of the text (default Helvetica)
- **fontsize** (*int*) – fontsize of the text (default 15)
- **action** (*function*) – action to take when button is pressed executed when the button is pressed (default None) the function should have no arguments
- **xy_anchor** (*str*) – specifies where x and y are relative to possible values are (default: sw): `nw n ne w c e sw s se`
- **env** (*Environment*) – environment where the component is defined if omitted, default_env will be used

Note: All measures are in screen coordinates On Pythonista, this functionality is emulated by salabim On other platforms, the tkinter functionality is used.

remove()

removes the button object. the ui object is removed from the ui queue, so effectively ending this ui

class salabim.**AnimateCircle**(*radius=None*, *radius1=None*, *arc_angle0=None*, *arc_angle1=None*, *draw_arc=None*, *x=None*, *y=None*, *fillcolor=None*, *linecolor=None*, *linewidth=None*, *text=None*, *fontsize=None*, *textcolor=None*, *font=None*, *angle=None*, *xy_anchor=None*, *layer=None*, *max_lines=None*, *offsetx=None*, *offsety=None*, *text_anchor=None*, *text_offsetx=None*, *text_offsety=None*, *arg=None*, *parent=None*, *visible=None*, *keep=None*, *env=None*, *screen_coordinates=False*, *over3d=None*)

Displays a (partial) circle or (partial) ellipse , optionally with a text

Parameters

- **radius** (*float*) – radius of the circle
- **radius1** (*float*) – the 'height' of the ellipse. If None (default), a circle will be drawn
- **arc_angle0** (*float*) – start angle of the circle (default 0)
- **arc_angle1** (*float*) – end angle of the circle (default 360) when arc_angle1 > arc_angle0 + 360, only 360 degrees will be shown
- **draw_arc** (*bool*) – if False (default), no arcs will be drawn if True, the arcs from and to the center will be drawn
- **x** (*float*) – position of anchor point (default 0)
- **y** (*float*) – position of anchor point (default 0)
- **xy_anchor** (*str*) – specifies where x and y are relative to possible values are (default: sw) : `nw n ne w c e sw s se` If null string, the given coordimates are used untranslated The positions corresponds to a full circle even if arc_angle0 and/or arc_angle1 are specified.

- **offsetx** (*float*) – offsets the x-coordinate of the circle (default 0)
- **offsety** (*float*) – offsets the y-coordinate of the circle (default 0)
- **linewidth** (*float*) – linewidth of the contour default 1
- **fillcolor** (*colorspec*) – color of interior (default foreground_color) default transparent
- **linecolor** (*colorspec*) – color of the contour (default transparent)
- **angle** (*float*) – angle of the circle/ellipse and/or text (in degrees) default: 0
- **text** (*str, tuple or list*) – the text to be displayed if text is str, the text may contain linefeeds, which are shown as individual lines
- **max_lines** (*int*) – the maximum of lines of text to be displayed if positive, it refers to the first max_lines lines if negative, it refers to the last -max_lines lines if zero (default), all lines will be displayed
- **font** (*str or list/tuple*) – font to be used for texts Either a string or a list/tuple of fontnames. If not found, uses calibri or arial
- **text_anchor** (*str*) – anchor position of text|n| specifies where to texts relative to the polygon point possible values are (default: c): `nw n ne w c e sw s se`
- **textcolor** (*colorspec*) – color of the text (default foreground_color)
- **text_offsetx** (*float*) – extra x offset to the text_anchor point
- **text_offsety** (*float*) – extra y offset to the text_anchor point
- **fontsize** (*float*) – fontsize of text (default 15)
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)
- **parent** (`Component`) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **screen_coordinates** (*bool*) – use screen_coordinates normally, the scale parameters are use for positioning and scaling objects. if True, screen_coordinates will be used instead.
- **over3d** (*bool*) – if True, this object will be rendered to the OpenGL window if False (default), the normal 2D plane will be used.

Note: All measures are in screen coordinates

All parameters, apart from parent, arg and env can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: title - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

class salabim.**AnimateCombined**(*animation_objects*, ***kwargs*)
Combines several Animate? objects

Parameters

- **animation_objects** (*iterable*) – iterable of Animate2dBase, Animate3dBase or AnimateCombined objects
- ****kwargs** (*dict*) – attributes to be set for objects in animation_objects

Notes

When an attribute of an AnimateCombined is assigned, it will propagate to all members, provided it has already that attribute. When an attribute of an AnimateCombined is queried, the value of the attribute of the first animation_object of the list that has such an attribute will be returned. If the attribute does not exist in any animation_object of the list, an AttributeError will be raised.

It is possible to use animation_objects with

```
an = sim.AnimationCombined(car.animation_objects[2:])
an = sim.AnimationCombined(car.animation3d_objects[3:])
```

append(*item*)
Add Animate2dBase, Animate3dBase or AnimateCombined object

Parameters **item** (*Animate2dBase, Animate3dBase or AnimateCombined*) – to be added

remove()
remove all members from the animation

show()
show all members in the animation

update(***kwargs*)
Updated one or more attributes

Parameters ****kwargs** (*dict*) – attributes to be set

class salabim.**AnimateEntry**(*x=0*, *y=0*, *number_of_chars=20*, *value=”*, *fillcolor=’fg’*, *color=’bg’*, *text=”*, *action=None*, *env=None*, *xy_anchor=’sw’*)
defines a button

Parameters

- **x** (*int*) – x-coordinate of centre of the button in screen coordinates (default 0)

- **y** (*int*) – y-coordinate of centre of the button in screen coordinates (default 0)
- **number_of_chars** (*int*) – number of characters displayed in the entry field (default 20)
- **fillcolor** (*colorspec*) – color of the entry background (default foreground_color)
- **color** (*colorspec*) – color of the text (default background_color)
- **value** (*str*) – initial value of the text of the entry (default null string)
- **action** (*function*) – action to take when the Enter-key is pressed the function should have no arguments
- **xy_anchor** (*str*) – specifies where x and y are relative to possible values are (default: sw): `nw n ne w c e sw s se`
- **env** (*Environment*) – environment where the component is defined if omitted, default_env will be used

Note: All measures are in screen coordinates This class is not available under Pythonista.

get()
get the current value of the entry

Returns **Current value of the entry**

Return type str

remove()
removes the entry object. the ui object is removed from the ui queue, so effectively ending this ui

class salabim.**AnimateImage**(*image=None, x=None, y=None, width=None, text=None, fontsize=None, textcolor=None, font=None, angle=None, alpha=None, xy_anchor=None, layer=None, max_lines=None, offsetx=None, offsety=None, text_anchor=None, text_offsetx=None, text_offsety=None, anchor=None, visible=None, keep=None, env=None, arg=None, screen_coordinates=False, over3d=None, parent=None*)
Displays an image, optionally with a text

Parameters

- **image** (*str, pathlib.Path or PIL Image*) – image to be displayed if used as function or method or in direct assigmnent, the image should be a file containing an image or a PIL image
- **x** (*float*) – position of anchor point (default 0)
- **y** (*float*) – position of anchor point (default 0)
- **xy_anchor** (*str*) – specifies where x and y are relative to possible values are (default: sw) : `nw n ne w c e sw s se` If null string, the given coordimates are used untranslated

- **anchor** (*str*) – specifies where the x and refer to possible values are (default: sw) : `nw n ne w c e sw s se`
- **offsetx** (*float*) – offsets the x-coordinate of the circle (default 0)
- **offsety** (*float*) – offsets the y-coordinate of the circle (default 0)
- **angle** (*float*) – angle of the image (in degrees) (default 0)
- **alpha** (*float*) – alpha of the image (0-255) (default 255)
- **width** (*float*) – width of the image (default: None = no scaling)
- **text** (*str, tuple or list*) – the text to be displayed if text is str, the text may contain linefeeds, which are shown as individual lines
- **max_lines** (*int*) – the maximum of lines of text to be displayed if positive, it refers to the first max_lines lines if negative, it refers to the last -max_lines lines if zero (default), all lines will be displayed
- **font** (*str or list/tuple*) – font to be used for texts Either a string or a list/tuple of fontnames. If not found, uses calibri or arial
- **text_anchor** (*str*) – anchor position of text|n| specifies where to texts relative to the polygon point possible values are (default: c): `nw n ne w c e sw s se`
- **textcolor** (*colorspec*) – color of the text (default foreground_color)
- **text_offsetx** (*float*) – extra x offset to the text_anchor point
- **text_offsety** (*float*) – extra y offset to the text_anchor point
- **fontsize** (*float*) – fontsize of text (default 15)
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)
- **parent** (*Component*) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **screen_coordinates** (*bool*) – use screen_coordinates normally, the scale parameters are used for positioning and scaling objects. if True, screen_coordinates will be used instead.
- **over3d** (*bool*) – if True, this object will be rendered to the OpenGL window if False (default), the normal 2D plane will be used.

Note: All measures are in screen coordinates

All parameters, apart from parent, arg and env can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: title - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

---

class salabim.**AnimateLine**(*spec=None*, *x=None*, *y=None*, *linecolor=None*, *linewidth=None*, *text=None*, *fontsize=None*, *textcolor=None*, *font=None*, *angle=None*, *xy_anchor=None*, *layer=None*, *max_lines=None*, *offsetx=None*, *offsety=None*, *as_points=None*, *text_anchor=None*, *text_offsetx=None*, *text_offsety=None*, *arg=None*, *parent=None*, *visible=None*, *keep=None*, *env=None*, *screen_coordinates=False*, *over3d=None*)

Displays a line, optionally with a text

Parameters

- **spec** (*tuple or list*) – should specify x0, y0, x1, y1, …
- **x** (*float*) – position of anchor point (default 0)
- **y** (*float*) – position of anchor point (default 0)
- **xy_anchor** (*str*) – specifies where x and y are relative to possible values are (default: sw) : `nw n ne` `w c e` `sw s se` If null string, the given coordimates are used untranslated
- **offsetx** (*float*) – offsets the x-coordinate of the line (default 0)
- **offsety** (*float*) – offsets the y-coordinate of the line (default 0)
- **linewidth** (*float*) – linewidth of the contour default 1
- **linecolor** (*colorspec*) – color of the contour (default foreground_color)
- **angle** (*float*) – angle of the line (in degrees) default: 0
- **as_points** (*bool*) – if False (default), the contour lines are drawn if True, only the corner points are shown
- **text** (*str, tuple or list*) – the text to be displayed if text is str, the text may contain linefeeds, which are shown as individual lines
- **max_lines** (*int*) – the maximum of lines of text to be displayed if positive, it refers to the first max_lines lines if negative, it refers to the last -max_lines lines if zero (default), all lines will be displayed
- **font** (*str or list/tuple*) – font to be used for texts Either a string or a list/tuple of fontnames. If not found, uses calibri or arial
- **text_anchor** (*str*) – anchor position of text|n| specifies where to texts relative to the polygon point possible values are (default: c): `nw n ne` `w c e` `sw s se`
- **textcolor** (*colorspec*) – color of the text (default foreground_color)
- **text_offsetx** (*float*) – extra x offset to the text_anchor point
- **text_offsety** (*float*) – extra y offset to the text_anchor point

- **fontsize** (*float*) – fontsize of text (default 15)
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)
- **parent** (Component) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **screen_coordinates** (*bool*) – use screen_coordinates normally, the scale parameters are use for positioning and scaling objects. if True, screen_coordinates will be used instead.
- **over3d** (*bool*) – if True, this object will be rendered to the OpenGL window if False (default), the normal 2D plane will be used.

Note: All measures are in screen coordinates

All parameters, apart from parent, arg and env can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: title - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

class salabim.**AnimateMonitor**(*monitor, linecolor='fg', linewidth=None, fillcolor='', bordercolor='fg', borderlinewidth=1, titlecolor='fg', nowcolor='red', titlefont='', titlefontsize=15, title=None, x=0, y=0, offsetx=0, offsety=0, angle=0, vertical_offset=0, parent=None, vertical_scale=5, horizontal_scale=1, width=200, height=75, xy_anchor='sw', vertical_map=<class 'float'>, labels=(), label_color='fg', label_font='', label_fontsize=15, label_anchor='e', label_offsetx=0, label_offsety=0, label_linewidth=1, label_linecolor='fg', as_points=None, over3d=None, layer=0, visible=True, keep=True, screen_coordinates=True, arg=None*)

animates a monitor in a panel

Parameters

- **monitor** (Monitor) – monitor to be animated
- **linecolor** (*colorspec*) – color of the line or points (default foreground color)
- **linewidth** (*int*) – width of the line or points (default 1 for level, 3 for non level monitors)
- **fillcolor** (*colorspec*) – color of the panel (default transparent)
- **bordercolor** (*colorspec*) – color of the border (default foreground color)
- **borderlinewidth** (*int*) – width of the line around the panel (default 1)
- **nowcolor** (*colorspec*) – color of the line indicating now (default red)
- **titlecolor** (*colorspec*) – color of the title (default foreground color)

- **titlefont** (*font*) – font of the title (default null string)
- **titlefontsize** (*int*) – size of the font of the title (default 15)
- **title** (*str*) – title to be shown above panel default: name of the monitor
- **x** (*int*) – x-coordinate of panel, relative to xy_anchor, default 0
- **y** (*int*) – y-coordinate of panel, relative to xy_anchor. default 0
- **offsetx** (*float*) – offsets the x-coordinate of the panel (default 0)
- **offsety** (*float*) – offsets the y-coordinate of the panel (default 0)
- **angle** (*float*) – rotation angle in degrees, default 0
- **xy_anchor** (*str*) – specifies where x and y are relative to possible values are (default: sw): `nw n ne` `w c e` `sw s se`
- **vertical_offset** (*float*) –

  the vertical position of x within the panel is vertical_offset + x * vertical_scale (default 0)
- **vertical_scale** (*float*) – the vertical position of x within the panel is vertical_offset + x * vertical_scale (default 5)
- **horizontal_scale** (*float*) – the relative horizontal position of time t within the panel is on t * horizontal_scale, possibly shifted (default 1)|n|
- **width** (*int*) – width of the panel (default 200)
- **height** (*int*) – height of the panel (default 75)
- **vertical_map** (*function*) – when a y-value has to be plotted it will be translated by this function default: float when the function results in a TypeError or ValueError, the value 0 is assumed when y-values are non numeric, it is advised to provide an approriate map function, like: vertical_map = "unknown red green blue yellow".split().index
- **labels** (*iterable*) – labels to be shown on the vertical axis (default: empty tuple) the placement of the labels is controlled by the vertical_map method
- **label_color** (*colorspec*) – color of labels (default: foreground color)
- **label_font** (*font*) – font of the labels (default null string)
- **label_fontsize** (*int*) – size of the font of the labels (default 15)
- **label_anchor** (*str*) – specifies where the label coordinates (as returned by map_value) are relative to possible values are (default: e): `nw n ne` `w c e` `sw s se`
- **label_offsetx** (*float*) – offsets the x-coordinate of the label (default 0)

- **label_offsety** (*float*) – offsets the y-coordinate of the label (default 0)
- **label_linewidth** (*int*) – width of the label line (default 1)
- **label_linecolor** (*colorspec*) – color of the label lines (default foreground color)
- **layer** (*int*) – layer (default 0)
- **as_points** (*bool*) – allows to override the line/point setting, which is by default False for level monitors and True for non level monitors
- **parent** (`Component`) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **over3d** (*bool*) – if True, this object will be rendered to the OpenGL window if False (default), the normal 2D plane will be used.
- **visible** (*bool*) – visible if False, animation monitor is not shown, shown otherwise (default True)
- **screen_coordinates** (*bool*) – use screen_coordinates if False, the scale parameters are use for positioning and scaling objects. if True (default), screen_coordinates will be used.

Note: All measures are in screen coordinates

monitor()

Returns **monitor this animation object refers**

Return type *Monitor*

remove()
removes the animate object and thus closes this animation

show(*unremove*)
It is possible to use this method if already shown

class salabim.**AnimatePoints**(*spec=None*, *x=None*, *y=None*, *linecolor=None*, *linewidth=None*, *text=None*, *fontsize=None*, *textcolor=None*, *font=None*, *angle=None*, *xy_anchor=None*, *layer=None*, *max_lines=None*, *offsetx=None*, *offsety=None*, *as_points=None*, *text_anchor=None*, *text_offsetx=None*, *text_offsety=None*, *arg=None*, *parent=None*, *visible=None*, *keep=None*, *env=None*, *screen_coordinates=False*, *over3d=None*)
Displays a series of points, optionally with a text

Parameters

- **spec** (*tuple or list*) – should specify x0, y0, x1, y1, . . .
- **x** (*float*) – position of anchor point (default 0)

- **y** (*float*) – position of anchor point (default 0)
- **xy_anchor** (*str*) – specifies where x and y are relative to possible values are (default: sw) : `nw n ne w c e sw s se` If null string, the given coordimates are used untranslated
- **offsetx** (*float*) – offsets the x-coordinate of the points (default 0)
- **offsety** (*float*) – offsets the y-coordinate of the points (default 0)
- **linewidth** (*float*) – width of the points default 1
- **linecolor** (*colorspec*) – color of the points (default foreground_color)
- **angle** (*float*) – angle of the points (in degrees) default: 0
- **as_points** (*bool*) – if False (default), the contour lines are drawn if True, only the corner points are shown
- **text** (*str, tuple or list*) – the text to be displayed if text is str, the text may contain linefeeds, which are shown as individual lines
- **max_lines** (*int*) – the maximum of lines of text to be displayed if positive, it refers to the first max_lines lines if negative, it refers to the last -max_lines lines if zero (default), all lines will be displayed
- **font** (*str or list/tuple*) – font to be used for texts Either a string or a list/tuple of fontnames. If not found, uses calibri or arial
- **text_anchor** (*str*) – anchor position of text|n| specifies where to texts relative to the polygon point possible values are (default: c): `nw n ne w c e sw s se`
- **textcolor** (*colorspec*) – color of the text (default foreground_color)
- **text_offsetx** (*float*) – extra x offset to the text_anchor point
- **text_offsety** (*float*) – extra y offset to the text_anchor point
- **fontsize** (*float*) – fontsize of text (default 15)
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)
- **parent** (`Component`) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **screen_coordinates** (*bool*) – use screen_coordinates normally, the scale parameters are use for positioning and scaling objects. if True, screen_coordinates will be used instead.
- **over3d** (*bool*) – if True, this object will be rendered to the OpenGL window if False (default), the normal 2D plane will be used.

Note: All measures are in screen coordinates

All parameters, apart from parent, arg and env can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: title - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

class salabim.**AnimatePolygon**(*spec=None*, *x=None*, *y=None*, *fillcolor=None*, *linecolor=None*, *linewidth=None*, *text=None*, *fontsize=None*, *textcolor=None*, *font=None*, *angle=None*, *xy_anchor=None*, *layer=None*, *max_lines=None*, *offsetx=None*, *offsety=None*, *as_points=None*, *text_anchor=None*, *text_offsetx=None*, *text_offsety=None*, *arg=None*, *parent=None*, *visible=None*, *keep=None*, *env=None*, *screen_coordinates=False*, *over3d=None*)

Displays a polygon, optionally with a text

Parameters

- **spec** (*tuple or list*) – should specify x0, y0, x1, y1, . . .
- **x** (*float*) – position of anchor point (default 0)
- **y** (*float*) – position of anchor point (default 0)
- **xy_anchor** (*str*) – specifies where x and y are relative to possible values are (default: sw) : `nw n ne w c e sw s se` If null string, the given coordinates are used untranslated
- **offsetx** (*float*) – offsets the x-coordinate of the polygon (default 0)
- **offsety** (*float*) – offsets the y-coordinate of the polygon (default 0)
- **linewidth** (*float*) – linewidth of the contour default 1
- **fillcolor** (*colorspec*) – color of interior (default foreground_color) default transparent
- **linecolor** (*colorspec*) – color of the contour (default transparent)
- **angle** (*float*) – angle of the polygon (in degrees) default: 0
- **as_points** (*bool*) – if False (default), the contour lines are drawn if True, only the corner points are shown
- **text** (*str, tuple or list*) – the text to be displayed if text is str, the text may contain linefeeds, which are shown as individual lines
- **max_lines** (*int*) – the maximum of lines of text to be displayed if positive, it refers to the first max_lines lines if negative, it refers to the last -max_lines lines if zero (default), all lines will be displayed
- **font** (*str or list/tuple*) – font to be used for texts Either a string or a list/tuple of fontnames. If not found, uses calibri or arial

- **text_anchor** (*str*) – anchor position of text|n| specifies where to texts relative to the polygon point possible values are (default: c): `nw n ne w c e sw s se`
- **textcolor** (*colorspec*) – color of the text (default foreground_color)
- **text_offsetx** (*float*) – extra x offset to the text_anchor point
- **text_offsety** (*float*) – extra y offset to the text_anchor point
- **fontsize** (*float*) – fontsize of text (default 15)
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)
- **parent** (Component) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **screen_coordinates** (*bool*) – use screen_coordinates normally, the scale parameters are use for positioning and scaling objects. if True, screen_coordinates will be used instead.
- **over3d** (*bool*) – if True, this object will be rendered to the OpenGL window if False (default), the normal 2D plane will be used.

Note: All measures are in screen coordinates

All parameters, apart from parent, arg and env can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: title - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

class salabim.**AnimateQueue**(*queue*, *x=50*, *y=50*, *direction='w'*, *trajectory=None*, *max_length=None*, *xy_anchor='sw'*, *reverse=False*, *title=None*, *titlecolor='fg'*, *titlefontsize=15*, *titlefont=''*, *titleoffsetx=None*, *titleoffsety=None*, *layer=0*, *id=None*, *arg=None*, *parent=None*, *over3d=None*, *keep=True*, *visible=True*)

Animates the component in a queue.

queue : Queue

x [float] x-position of the first component in the queue default: 50

y [float] y-position of the first component in the queue default: 50

direction [str] if "w", waiting line runs westwards (i.e. from right to left) if "n", waiting line runs northeards (i.e. from bottom to top) if "e", waiting line runs eastwards (i.e. from left to right) (default) if "s", waiting line runs southwards (i.e. from top to bottom)

:

trajectory [Trajectory] trajectory to be followed. Overrides any given directory

reverse [bool] if False (default), display in normal order. If True, reversed.

max_length [int] maximum number of components to be displayed

xy_anchor [str] specifies where x and y are relative to possible values are (default: sw): `nw n ne` `w c e` `sw s se`

titlecolor [colorspec] color of the title (default foreground color)

titlefont [font] font of the title (default null string)

titlefontsize [int] size of the font of the title (default 15)

title [str] title to be shown above queue default: name of the queue

titleoffsetx [float] x-offset of the title relative to the start of the queue default: 25 if direction is w, -25 otherwise

titleoffsety [float] y-offset of the title relative to the start of the queue default: -25 if direction is s, -25 otherwise

id [any] the animation works by calling the animation_objects method of each component, optionally with id. By default, this is self, but can be overriden, particularly with the queue

arg [any] this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)

visible [bool] if False, nothing will be shown (default True)

keep [bool] if False, animation object will be taken from the animation objects. With show(), the animation can be reshown. (default True)

parent [Component] component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible

All measures are in screen coordinates

All parameters, apart from queue, id, arg and parent can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: title - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

show(*unremove*)
It is possible to use this method if already shown

class salabim.**AnimateRectangle**(*spec=None*, *x=None*, *y=None*, *fillcolor=None*, *linecolor=None*, *linewidth=None*, *text=None*, *fontsize=None*, *textcolor=None*, *font=None*, *angle=None*, *xy_anchor=None*, *layer=None*, *max_lines=None*, *offsetx=None*, *offsety=None*, *as_points=None*, *text_anchor=None*, *text_offsetx=None*, *text_offsety=None*, *arg=None*, *parent=None*, *visible=None*, *keep=None*, *env=None*, *screen_coordinates=False*, *over3d=None*)
Displays a rectangle, optionally with a text

Parameters

- **spec** (*four item tuple or list*) – should specify xlowerleft, ylowerleft, xupperright, yupperright
- **x** (*float*) – position of anchor point (default 0)
- **y** (*float*) – position of anchor point (default 0)
- **xy_anchor** (*str*) – specifies where x and y are relative to possible values are (default: sw) : `nw n ne w c e sw s se` If null string, the given coordimates are used untranslated
- **offsetx** (*float*) – offsets the x-coordinate of the rectangle (default 0)
- **offsety** (*float*) – offsets the y-coordinate of the rectangle (default 0)
- **linewidth** (*float*) – linewidth of the contour default 1
- **fillcolor** (*colorspec*) – color of interior (default foreground_color) default transparent
- **linecolor** (*colorspec*) – color of the contour (default transparent)
- **angle** (*float*) – angle of the rectangle (in degrees) default: 0
- **as_points** (*bool*) – if False (default), the contour lines are drawn if True, only the corner points are shown
- **text** (*str, tuple or list*) – the text to be displayed if text is str, the text may contain linefeeds, which are shown as individual lines
- **max_lines** (*int*) – the maximum of lines of text to be displayed if positive, it refers to the first max_lines lines if negative, it refers to the last -max_lines lines if zero (default), all lines will be displayed
- **font** (*str or list/tuple*) – font to be used for texts Either a string or a list/tuple of fontnames. If not found, uses calibri or arial
- **text_anchor** (*str*) – anchor position of text|n| specifies where to texts relative to the rectangle point possible values are (default: c): `nw n ne w c e sw s se`
- **textcolor** (*colorspec*) – color of the text (default foreground_color)
- **text_offsetx** (*float*) – extra x offset to the text_anchor point
- **text_offsety** (*float*) – extra y offset to the text_anchor point
- **fontsize** (*float*) – fontsize of text (default 15)
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)
- **parent** (*Component*) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible

Note: All measures are in screen coordinates

All parameters, apart from parent, arg and env can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: title - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

class salabim.**AnimateSlider**(*x=0, y=0, width=100, height=20, vmin=0, vmax=10, v=None, resolution=1, background_color='bg', foreground_color='fg', trough_color='lightgray', show_value=True, label='', font='', fontsize=12, action=None, xy_anchor='sw', env=None, linecolor=None, labelcolor=None, layer=None*)

defines a slider

Parameters

- **x** (*int*) – x-coordinate of centre of the slider in screen coordinates (default 0)
- **y** (*int*) – y-coordinate of centre of the slider in screen coordinates (default 0)
- **vmin** (*float*) – minimum value of the slider (default 0)
- **vmax** (*float*) – maximum value of the slider (default 0)
- **v** (*float*) – initial value of the slider (default 0) should be between vmin and vmax
- **resolution** (*float*) – step size of value (default 1)
- **width** (*float*) – width of slider in screen coordinates (default 100)
- **height** (*float*) – height of slider in screen coordinates (default 20)
- **foreground_color** (*colorspec*) – color of the foreground (default "fg")
- **background_color** (*colorspec*) – color of the backgroundground (default "bg")
- **trough_color** (*colorspec*) – color of the trough (default "lightgrey")
- **show_value** (*boolean*) – if True (default), show values; if False don't show values
- **label** (*str*) – label if the slider (default null string)
- **font** (*str*) – font of the text (default Helvetica)
- **fontsize** (*int*) – fontsize of the text (default 12)
- **action** (*function*) – function executed when the slider value is changed (default None) the function should have one argument, being the new value if None (default), no action

- **xy_anchor** (*str*) – specifies where x and y are relative to possible values are (default: sw): `nw n ne w c e sw s se`
- **env** (`Environment`) – environment where the component is defined if omitted, default_env will be used

Note: The current value of the slider is the v attibute of the slider. All measures are in screen coordinates On Pythonista, this functionality is emulated by salabim On other platforms, the tkinter functionality is used.

remove()

removes the slider object The ui object is removed from the ui queue, so effectively ending this ui

v(*value=None*)

value

Parameters value (*float*) – new value if omitted, no change

Returns Current value of the slider

Return type float

class salabim.**AnimateText**(*text=None, x=None, y=None, font=None, fontsize=None, textcolor=None, text_anchor=None, angle=None, xy_anchor=None, layer=None, max_lines=None, offsetx=None, offsety=None, arg=None, visible=None, keep=None, parent=None, env=None, screen_coordinates=False, over3d=None*)

Displays a text

Parameters

- **text** (*str, tuple or list*) – the text to be displayed if text is str, the text may contain linefeeds, which are shown as individual lines if text is tple or list, each item is displayed on a separate line
- **x** (*float*) – position of anchor point (default 0)
- **y** (*float*) – position of anchor point (default 0)
- **xy_anchor** (*str*) – specifies where x and y are relative to possible values are (default: sw) : `nw n ne w c e sw s se` If null string, the given coordimates are used untranslated
- **offsetx** (*float*) – offsets the x-coordinate of the rectangle (default 0)
- **offsety** (*float*) – offsets the y-coordinate of the rectangle (default 0)
- **angle** (*float*) – angle of the text (in degrees) default: 0
- **max_lines** (*int*) – the maximum of lines of text to be displayed if positive, it refers to the first max_lines lines if negative, it refers to the last -max_lines lines if zero (default), all lines will be displayed

- **font** (*str or list/tuple*) – font to be used for texts Either a string or a list/tuple of fontnames. If not found, uses calibri or arial
- **text_anchor** (*str*) – anchor position of text|n| specifies where to texts relative to the rectangle point possible values are (default: c): `nw n ne w c e sw s se`
- **textcolor** (*colorspec*) – color of the text (default foreground_color)
- **fontsize** (*float*) – fontsize of text (default 15)
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)
- **parent** (*Component*) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **screen_coordinates** (*bool*) – use screen_coordinates normally, the scale parameters are use for positioning and scaling objects. if True, screen_coordinates will be used instead.
- **over3d** (*bool*) – if True, this object will be rendered to the OpenGL window if False (default), the normal 2D plane will be used.

Note: All measures are in screen coordinates

All parameters, apart from parent, arg and env can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: title - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

14.3 Component

class salabim.**Component**(*name=None*, *at=None*, *delay=None*, *priority=None*, *urgent=None*, *process=None*, *suppress_trace=False*, *suppress_pause_at_step=False*, *skip_standby=False*, *mode=”*, *cap_now=None*, *env=None*, ***kwargs*)

Component object

A salabim component is used as component (primarily for queueing) or as a component with a process Usually, a component will be defined as a subclass of Component.

Parameters

- **name** (*str*) – name of the component. if the name ends with a period (.), auto serializing will be applied if the name end with a comma, auto serializing starting at 1 will be applied if omitted, the name will be derived from the class it is defined in (lowercased)
- **at** (*float or distribution*) – schedule time if omitted, now is used if distribution, the distribution is sampled
- **delay** (*float or distributiom*) – schedule with a delay if omitted, no delay if distribution, the distribution is sampled

- **priority** (*float*) – priority default: 0 if a component has the same time on the event list, this component is sorted accoring to the priority.
- **urgent** (*bool*) – urgency indicator if False (default), the component will be scheduled behind all other components scheduled for the same time and priority if True, the component will be scheduled in front of all components scheduled for the same time and priority
- **process** (*str*) – name of process to be started. if None (default), it will try to start self.process() if null string, no process will be started even if self.process() exists, i.e. become a data component.
- **suppress_trace** (*bool*) – suppress_trace indicator if True, this component will be excluded from the trace If False (default), the component will be traced Can be queried or set later with the suppress_trace method.
- **suppress_pause_at_step** (*bool*) – suppress_pause_at_step indicator if True, if this component becomes current, do not pause when stepping If False (default), the component will be paused when stepping Can be queried or set later with the suppress_pause_at_step method.
- **skip_standby** (*bool*) – skip_standby indicator if True, after this component became current, do not activate standby components If False (default), after the component became current activate standby components Can be queried or set later with the skip_standby method.
- **mode** (*str preferred*) – mode will be used in trace and can be used in animations if omitted, the mode will be "". also mode_time will be set to now.
- **cap_now** (*bool*) – indicator whether times (at, delay) in the past are allowed. If, so now() will be used. default: sys.default_cap_now(), usualy False
- **env** (*Environment*) – environment where the component is defined if omitted, default_env will be used

activate (*at=None*, *delay=0*, *priority=0*, *urgent=False*, *process=None*, *keep_request=False*, *keep_wait=False*, *mode=None*, *cap_now=None*, ***kwargs*)
activate component

Parameters

- **at** (*float or distribution*) – schedule time if omitted, now is used inf is allowed if distribution, the distribution is sampled
- **delay** (*float or distribution*) – schedule with a delay if omitted, no delay if distribution, the distribution is sampled
- **priority** (*float*) – priority default: 0 if a component has the same time on the event list, this component is sorted accoring to the priority.
- **urgent** (*bool*) – urgency indicator if False (default), the component will be scheduled behind all other components scheduled for the same time and priority if True, the component will be scheduled in front of all components scheduled for the same time and priority
- **process** (*str*) – name of process to be started. if None (default), process will not be changed if the component is a data component, the generator function process will be used as the default process. note that the function *must* be a generator, i.e. contains at least one yield.
- **keep_request** (*bool*) – this affects only components that are requesting. if True, the requests will be kept and thus the status will remain requesting if False (the default), the request(s) will be canceled and the status will become scheduled

- **keep_wait** (*bool*) – this affects only components that are waiting. if True, the waits will be kept and thus the status will remain waiting if False (the default), the wait(s) will be canceled and the status will become scheduled
- **cap_now** (*bool*) – indicator whether times (at, delay) in the past are allowed. If, so now() will be used. default: sys.default_cap_now(), usualy False
- **mode** (*str preferred*) – mode will be used in the trace and can be used in animations if nothing specified, the mode will be unchanged. also mode_time will be set to now, if mode is set.

Note: if to be applied to the current component, use `yield self.activate()`. if both at and delay are specified, the component becomes current at the sum of the two values.

animation3d_objects(*id*)

defines how to display a component in Animate3dQueue

Parameters **id** (*any*) – id as given by Animate3dQueue. Note that by default this the reference to the Animate3dQueue object.

Returns size_x : how much to displace the next component in x-direction, if applicable size_y : how much to displace the next component in y-direction, if applicable size_z : how much to displace the next component in z-direction, if applicable animation objects : instances of Animate3dBase class default behaviour: white 3dbox of size 8, placed on the z=0 plane (displacements 10).

Return type List or tuple containg

Note: If you override this method, be sure to use the same header, either with or without the id parameter.

Note: The animation object should support the x_offset, y_offset and z_offset attributes, in order to be able to position the object correctly. All native salabim Animate3d classes are offset aware.

animation_objects(*id*)

defines how to display a component in AnimateQueue

Parameters **id** (*any*) – id as given by AnimateQueue. Note that by default this the reference to the AnimateQueue object.

Returns size_x : how much to displace the next component in x-direction, if applicable size_y : how much to displace the next component in y-direction, if applicable animation objects : instances of Animate class default behaviour: square of size 40 (displacements 50), with the sequence number centered.

Return type List or tuple containg

Note: If you override this method, be sure to use the same header, either with or without the id parameter.

base_name()

Returns **base name of the component (the name used at initialization)**

Return type str

cancel(*mode=None*)
cancel component (makes the component data)

Parameters **mode** (*str preferred*) – mode will be used in trace and can be used in animations if nothing specified, the mode will be unchanged.
also mode_time will be set to now, if mode is set.

Note: if to be used for the current component, use `yield self.cancel()`.

claimed_quantity(*resource*)

Parameters **resource** (*Resoure*) – resource to be queried

Returns **the claimed quantity from a resource** – if the resource is not claimed, 0 will be returned

Return type float or int

claimed_resources()

Returns **list of claimed resources**

Return type list

count(*q=None*)
queue count

Parameters **q** (*Queue*) – queue to check or if omitted, the number of queues where the component is in

Returns **1 if component is in q, 0 otherwise** – if q is omitted, the number of queues where the component is in

Return type int

creation_time()

Returns **time the component was created**

Return type float

deregister(*registry*)

deregisters the component in the registry

Parameters **registry** (*list*) – list of registered components

Returns **component (self)**

Return type *Component*

enter(*q*)

enters a queue at the tail

Parameters **q** (Queue) – queue to enter

Note: the priority will be set to the priority of the tail component of the queue, if any or 0 if queue is empty

enter_at_head(*q*)

enters a queue at the head

Parameters **q** (Queue) – queue to enter

Note: the priority will be set to the priority of the head component of the queue, if any or 0 if queue is empty

enter_behind(*q, poscomponent*)

enters a queue behind a component

Parameters

- **q** (Queue) – queue to enter
- **poscomponent** (Component) – component to be entered behind

Note: the priority will be set to the priority of poscomponent

enter_in_front_of(*q, poscomponent*)

enters a queue in front of a component

Parameters

- **q** (`Queue`) – queue to enter
- **poscomponent** (`Component`) – component to be entered in front of

Note: the priority will be set to the priority of poscomponent

enter_sorted (*q*, *priority*)
enters a queue, according to the priority

Parameters

- **q** (`Queue`) – queue to enter
- **priority** (*type that can be compared with other priorities in the queue*) – priority in the queue

Note: The component is placed just before the first component with a priority > given priority

enter_time (*q*)

Parameters **q** (`Queue`) – queue where component belongs to

Returns **time the component entered the queue**

Return type float

failed ()

Returns

- **True, if the latest request/wait has failed (either by timeout or external)** (*bool*)
- *False, otherwise*

get (**args*, ***kwargs*)
equivalent to request

hold (*duration=None*, *till=None*, *priority=0*, *urgent=False*, *mode=None*, *cap_now=None*)
hold the component

Parameters

- **duration** (*float or distribution*) – specifies the duration if omitted, 0 is used inf is allowed if distribution, the distribution is sampled

- **till** (*float or distribution*) – specifies at what time the component will become current if omitted, now is used inf is allowed if distribution, the distribution is sampled
- **priority** (*float*) – priority default: 0 if a component has the same time on the event list, this component is sorted accoring to the priority.
- **urgent** (*bool*) – urgency indicator if False (default), the component will be scheduled behind all other components scheduled for the same time and priority if True, the component will be scheduled in front of all components scheduled for the same time and priority
- **mode** (*str preferred*) – mode will be used in trace and can be used in animations if nothing specified, the mode will be unchanged. also mode_time will be set to now, if mode is set.
- **cap_now** (*bool*) – indicator whether times (duration, till) in the past are allowed. If, so now() will be used. default: sys.default_cap_now(), usualy False

Note: if to be used for the current component, use `yield self.hold(...)`.

if both duration and till are specified, the component will become current at the sum of these two.

index(*q*)

Parameters **q** (Queue) – queue to be queried

Returns **index of component in q** – if component belongs to q -1 if component does not belong to q

Return type int

interrupt(*mode=None*)

interrupt the component

Parameters **mode** (*str preferred*) – mode will be used in trace and can be used in animations if nothing is specified, the mode will be unchanged. also mode_time will be set to now, if mode is set.

Note: Cannot be applied on the current component. Use resume() to resume

interrupt_level()

returns interrupt level of an interrupted component non interrupted components return 0

interrupted_status()

returns the original status of an interrupted component

possible values are

- passive
- scheduled
- requesting
- waiting
- standby

isbumped(*resource=None*)
check whether component is bumped from resource

Parameters **resource** (`Resource`) – resource to be checked if omitted, checks whether component belongs to any resource claimers

Returns **True if this component is not in the resource claimers** – False otherwise

Return type bool

isclaiming(*resource=None*)
check whether component is claiming from resource

Parameters **resource** (`Resource`) – resource to be checked if omitted, checks whether component is in any resource claimers

Returns **True if this component is in the resource claimers** – False otherwise

Return type bool

iscurrent()

Returns **True if status is current, False otherwise**

Return type bool

Note: Be sure to always include the parentheses, otherwise the result will be always True!

isdata()

Returns **True if status is data, False otherwise**

Return type bool

Note: Be sure to always include the parentheses, otherwise the result will be always True!

isinterrupted()

Returns **True if status is interrupted, False otherwise**

Return type bool

Note: Be sure to always include the parentheses, otherwise the result will be always True

ispassive()

Returns **True if status is passive, False otherwise**

Return type bool

Note: Be sure to always include the parentheses, otherwise the result will be always True!

isrequesting()

Returns **True if status is requesting, False otherwise**

Return type bool

Note: Be sure to always include the parentheses, otherwise the result will be always True!

isscheduled()

Returns **True if status is scheduled, False otherwise**

Return type bool

Note: Be sure to always include the parentheses, otherwise the result will be always True!

isstandby()

Returns **True if status is standby, False otherwise**

Return type bool

Note: Be sure to always include the parentheses, otherwise the result will be always True

iswaiting()

Returns **True if status is waiting, False otherwise**

Return type bool

Note: Be sure to always include the parentheses, otherwise the result will be always True!

leave(*q=None*)

leave queue

Parameters **q** (`Queue`) – queue to leave

Note: statistics are updated accordingly

line_number()

current line number of the process

Returns **Current line number** – for data components, "" will be returned

Return type str

mode_time()

Returns **time the component got it's latest mode** – For a new component this is the time the component was created. this function is particularly useful for animations.

Return type float

name(*value=None*)

Parameters **value** (*str*) – new name of the component if omitted, no change

Returns **Name of the component**

Return type str

Note: base_name and sequence_number are not affected if the name is changed

passivate (*mode=None*)
passivate the component

Parameters **mode** (*str preferred*) – mode will be used in trace and can be used in animations if nothing is specified, the mode will be unchanged. also mode_time will be set to now, if mode is set.

Note: if to be used for the current component (nearly always the case), use `yield self.passivate()`.

predecessor (*q*)

Parameters

- **q** (Queue) – queue where the component belongs to
- **Returns** (Component) – predecessor of the component in the queue if component is not at the head. returns None if component is at the head.

print_info (*as_str=False*, *file=None*)
prints information about the component

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns info (if as_str is True)

Return type str

priority (*q*, *priority=None*)
gets/sets the priority of a component in a queue

Parameters

- **q** (Queue) – queue where the component belongs to
- **priority** (*type that can be compared with other priorities in the queue*) – priority in queue if omitted, no change

Returns the priority of the component in the queue

Return type float

Note: if you change the priority, the order of the queue may change

put (**args*, ***kwargs*)
equivalent to request, but anonymous quantities are negated

queues ()

Returns **set of queues where the component belongs to**

Return type set

register (*registry*)
registers the component in the registry

Parameters **registry** (*list*) – list of (to be) registered objects

Returns **component (self)**

Return type *Component*

Note: Use Component.deregister if component does not longer need to be registered.

release (**args*)
release a quantity from a resource or resources

Parameters **args** (*sequence of items, where each items can be*) –

- a resources, where quantity=current claimed quantity
- a tuple/list containing a resource and the quantity to be released

Note: It is not possible to release from an anonymous resource, this way. Use Resource.release() in that case.

Example

yield self.request(r1,(r2,2),(r3,3,100)) –> requests 1 from r1, 2 from r2 and 3 from r3 with priority 100

c1.release –> releases 1 from r1, 2 from r2 and 3 from r3

yield self.request(r1,(r2,2),(r3,3,100)) c1.release((r2,1)) –> releases 1 from r2 yield self.request(r1,(r2,2),(r3,3,100)) c1.release((r2,1),r3) –> releases 2 from r2,and 3 from r3

remaining_duration(*value=None*, *priority=0*, *urgent=False*)

Parameters

- **value** (*float*) – set the remaining_duration The action depends on the status where the component is in: - passive: the remaining duration is update according to the given value - standby and current: not allowed - scheduled: the component is rescheduled according to the given value - waiting or requesting: the fail_at is set according to the given value - interrupted: the remaining_duration is updated according to the given value
- **priority** (*float*) – priority default: 0 if a component has the same time on the event list, this component is sorted accoring to the priority.
- **urgent** (*bool*) – urgency indicator if False (default), the component will be scheduled behind all other components scheduled for the same time and priority if True, the component will be scheduled in front of all components scheduled for the same time and priority

Returns **remaining duration** – if passive, remaining time at time of passivate if scheduled, remaing time till scheduled time if requesting or waiting, time till fail_at time else: 0

Return type float

Note: This method is useful for interrupting a process and then resuming it, after some (breakdown) time

remove_animation_children()

removes animation children

Note: Normally, the animation_children are removed automatically upon termination of a component (when it terminates)

request(**args*, ***kwargs*)

request from a resource or resources

Parameters

- **args** (*sequence of items where each item can be:*) –
  - resource, where quantity=1, priority=tail of requesters queue
  - **tuples/list containing a resource, a quantity and optionally a priority.** if the priority is not specified, the request for the resource be added to the tail of the requesters queue
- **priority** (*float*) – priority of the fail event|n| default: 0 if a component has the same time on the event list, this component is sorted accoring to the priority.

- **urgent** (*bool*) – urgency indicator if False (default), the component will be scheduled behind all other components scheduled for the same time and priority if True, the component will be scheduled in front of all components scheduled for the same time and priority
- **fail_at** (*float or distribution*) – time out if the request is not honored before fail_at, the request will be cancelled and the parameter failed will be set. if not specified, the request will not time out. if distribution, the distribution is sampled
- **fail_delay** (*float or distribution*) – time out if the request is not honored before now+fail_delay, the request will be cancelled and the parameter failed will be set. if not specified, the request will not time out. if distribution, the distribution is sampled
- **oneof** (*bool*) – if oneof is True, just one of the requests has to be met (or condition), where honoring follows the order given. if oneof is False (default), all requests have to be met to be honored
- **mode** (*str preferred*) – mode will be used in trace and can be used in animations if nothing specified, the mode will be unchanged. also mode_time will be set to now, if mode is set.
- **cap_now** (*bool*) – indicator whether times (fail_at, fail_delay) in the past are allowed. If, so now() will be used. default: sys.default_cap_now(), usualy False

Note: Not allowed for data components or main.

If to be used for the current component (which will be nearly always the case), use `yield self.request(...)`.

If the same resource is specified more that once, the quantities are summed

The requested quantity may exceed the current capacity of a resource

The parameter failed will be reset by a calling request or wait

Example

`yield self.request(r1)` –> requests 1 from r1 `yield self.request(r1,r2)` –> requests 1 from r1 and 1 from r2 `yield self.request(r1, (r2,2),(r3,3,100))` –> requests 1 from r1, 2 from r2 and 3 from r3 with priority 100 `yield self.request((r1,1),(r2,2))` –> requests 1 from r1, 2 from r2 `yield self.request(r1, r2, r3, oneoff=True)` –> requests 1 from r1, r2 or r3

requested_quantity(*resource*)

Parameters resource (*Resoure*) – resource to be queried

Returns the requested (not yet honored) quantity from a resource – if there is no request for the resource, 0 will be returned

Return type float or int

requested_resources()

Returns **list of requested resources**

Return type list

resume(*all=False*, *mode=None*, *priority=0*, *urgent=False*)
resumes an interrupted component

Parameters

- **all** (*bool*) – if True, the component returns to the original status, regardless of the number of interrupt levels if False (default), the interrupt level will be decremented and if the level reaches 0, the component will return to the original status.
- **mode** (*str preferred*) – mode will be used in trace and can be used in animations if nothing is specified, the mode will be unchanged. also mode_time will be set to now, if mode is set.
- **priority** (*float*) – priority default: 0 if a component has the same time on the event list, this component is sorted accoring to the priority.

urgent [bool] urgency indicator if False (default), the component will be scheduled behind all other components scheduled for the same time and priority if True, the component will be scheduled in front of all components scheduled for the same time and priority

Note: Can be only applied to interrupted components.

running_process()

Returns **name of the running process** – if data component, None

Return type str

scheduled_priority()

Returns **priority the component is scheduled with** – returns None otherwise

Return type float

Note: The method has to traverse the event list, so performance may be an issue.

scheduled_time()

Returns **time the component scheduled for, if it is scheduled** – returns inf otherwise

Return type float

sequence_number()

Returns sequence_number of the component – (the sequence number at initialization) normally this will be the integer value of a serialized name, but also non serialized names (without a dotcomma at the end) will be numbered)

Return type int

set_mode(*value=None*)

Parameters value (*any, str recommended*) – new mode mode_time will be set to now if omitted, no change

setup()

called immediately after initialization of a component.

by default this is a dummy method, but it can be overridden.

only keyword arguments will be passed

Example

class Car(sim.Component):

def setup(self, color): self.color = color

def process(self): ...

redcar=Car(color="red") bluecar=Car(color="blue")

skip_standby(*value=None*)

Parameters value (*bool*) – new skip_standby value if omitted, no change

Returns skip_standby indicator – components with the skip_standby indicator of True, will not activate standby components after the component became current.

Return type bool

standby(*mode=None*)

puts the component in standby mode

Parameters mode (*str preferred*) – mode will be used in trace and can be used in animations if nothing specified, the mode will be unchanged. also mode_time will be set to now, if mode is set.

Note: Not allowed for data components or main.

if to be used for the current component (which will be nearly always the case), use `yield self.standby()`.

---

status()

returns the status of a component

possible values are

- data
- passive
- scheduled
- requesting
- waiting
- current
- standby
- interrupted

successor(*q*)

Parameters **q** (`Queue`) – queue where the component belongs to

Returns **the successor of the component in the queue** – if component is not at the tail. returns None if component is at the tail.

Return type *Component*

suppress_pause_at_step(*value=None*)

Parameters **value** (*bool*) – new suppress_trace value if omitted, no change

Returns **suppress_pause_at_step** – components with the suppress_pause_at_step of True, will be ignored in a step

Return type bool

suppress_trace(*value=None*)

Parameters **value** (*bool*) – new suppress_trace value if omitted, no change

Returns **suppress_trace** – components with the suppress_status of True, will be ignored in the trace

Return type bool

wait(**args*, ***kwargs*)
wait for any or all of the given state values are met

Parameters

- **args** (*sequence of items, where each item can be*) –
  - a state, where value=True, priority=tail of waiters queue)
  - **a tuple/list containing** state, a value and optionally a priority. if the priority is not specified, this component will be added to the tail of the waiters queue
- **priority** (*float*) – priority of the fail event|n| default: 0 if a component has the same time on the event list, this component is sorted accoring to the priority.
- **urgent** (*bool*) – urgency indicator if False (default), the component will be scheduled behind all other components scheduled for the same time and priority if True, the component will be scheduled in front of all components scheduled for the same time and priority
- **fail_at** (*float or distribution*) – time out if the wait is not honored before fail_at, the wait will be cancelled and the parameter failed will be set. if not specified, the wait will not time out. if distribution, the distribution is sampled
- **fail_delay** (*float or distribution*) – time out if the wait is not honored before now+fail_delay, the request will be cancelled and the parameter failed will be set. if not specified, the wait will not time out. if distribution, the distribution is sampled
- **all** (*bool*) – if False (default), continue, if any of the given state/values is met if True, continue if all of the given state/values are met
- **mode** (*str preferred*) – mode will be used in trace and can be used in animations if nothing specified, the mode will be unchanged. also mode_time will be set to now, if mode is set.
- **cap_now** (*bool*) – indicator whether times (fail_at, fail_duration) in the past are allowed. If, so now() will be used. default: sys.default_cap_now(), usualy False

---

Note: Not allowed for data components or main.

If to be used for the current component (which will be nearly always the case), use `yield self.wait(...)`.

It is allowed to wait for more than one value of a state the parameter failed will be reset by a calling wait

If you want to check for all components to meet a value (and clause), use Component.wait(. . . , all=True)

The value may be specified in three different ways:

- constant, that value is just compared to state.value() yield self.wait((light,"red"))

- an expression, containg one or more $-signs the $ is replaced by state.value(), each time the condition is tested. self refers to the component under test, state refers to the state under test. yield self.wait((light,'$ in ("red","yellow")')) yield self.wait((level,"$<30"))
- a function. In that case the parameter should function that should accept three arguments: the value, the component under test and the state under test. usually the function will be a lambda function, but that's not a requirement. yield self.wait((light,lambda t, comp, state: t in ("red","yellow"))) yield self.wait((level,lambda t, comp, state: t < 30))

Example

`yield self.wait(s1)` –> waits for s1.value()==True `yield self.wait(s1,s2)` –> waits for s1.value()==True or s2.value==True `yield self.wait((s1,False,100),(s2,"on"),s3)` –> waits for s1.value()==False or s2.value=="on" or s3.value()==True s1 is at the tail of waiters, because of the set priority `yield self.wait(s1,s2,all=True)` –> waits for s1.value()==True and s2.value==True

14.4 ComponentGenerator

class salabim.**ComponentGenerator**(*component_class*, *generator_name=None*, *at=None*, *delay=None*, *till=None*, *duration=None*, *number=None*, *iat=None*, *force_at=False*, *force_till=False*, *suppress_trace=False*, *suppress_pause_at_step=False*, *disturbance=None*, *env=None*, ***kwargs*)

Component generator object

A component generator can be used to genetate components There are two ways of generating components: - according to a given inter arrival time (iat) value or distribution - random spread over a given time interval

Parameters

- **component_class** (*callable, usually a subclass of Component or Pdf or Cdf distribution*) – the type of components to be generated in case of a distribution, the Pdf or Cdf should return a callable
- **generator_name** (*str*) – name of the component generator. if the name ends with a period (.), auto serializing will be applied if the name end with a comma, auto serializing starting at 1 will be applied if omitted, the name will be derived from the name of the component_class, padded with '.generator'
- **at** (*float or distribution*) – time where the generator starts time if omitted, now is used if distribution, the distribution is sampled
- **delay** (*float or distribution*) – delay where the generator starts (at = now + delay) if omitted, no delay if distribution, the distribution is sampled
- **till** (*float or distribution*) – time up to which components should be generated if omitted, no end if distribution, the distribution is sampled

- **duration** (*float or distribution*) – duration to which components should be generated (till = now + duration) if omitted, no end if distribution, the distribution is sampled
- **number** (*int or distribution*) – (maximum) number of components to be generated if distribution, the distribution is sampled
- **iat** (*float or distribution*) – inter arrival time (distribution). if None (default), a random spread over the interval (at, till) will be used
- **force_at** (*bool*) –

  for iat generation: if False (default), the first component will be generated at time = at + sample from the iat if True, the first component will be generated at time = at

  for random spread generation: if False (default), no force for time = at if True, force the first generation at time = at
- **force_till** (*bool*) – only possible for random spread generation: if False (default), no force for time = till if True, force the last generated component at time = till
- **disturbance** (*callable (usually a distribution)*) – for each component to be generated, the disturbance call (sampling) is added to the actual generation time. disturbance may only be used together with iat. The force_at parameter is not allowed in that case.
- **suppress_trace** (*bool*) – suppress_trace indicator if True, the component generator events will be excluded from the trace If False (default), the component generator will be traced Can be queried or set later with the suppress_trace method.
- **suppress_pause_at_step** (*bool*) – suppress_pause_at_step indicator if True, if this component generator becomes current, do not pause when stepping If False (default), the component generator will be paused when stepping Can be queried or set later with the suppress_pause_at_step method.
- **cap_now** (*bool*) – indicator whether times (activation) in the past are allowed. If, so now() will be used. default: sys.default_cap_now(), usualy False
- **env** (*Environment*) – environment where the component is defined if omitted, default_env will be used

Note: For iat distributions: if till/duration and number are specified, the generation stops whichever condition comes first.

print_info(*as_str=False*, *file=None*)

prints information about the component generator

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

14.5 Distributions

class salabim.**_Expression**(*dis0*, *dis1*, *op*)

expression distribution

This class is only created when using an expression with one ore more distributions.

Note: The randomstream of the distribution(s) in the expression are used.

mean()

Returns **Mean of the expression of distribution(s)** – returns nan if mean can't be calculated

Return type float

print_info(*as_str=False*, *file=None*)

prints information about the expression of distribution(s)

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

sample()

Returns **Sample of the expression of distribution(s)**

Return type float

class salabim.**Beta**(*alpha*, *beta*, *randomstream=None*)

beta distribution

Parameters

- **alpha** (*float*) – alpha shape of the distribution should be >0
- **beta** (*float*) – beta shape of the distribution should be >0

- **randomstream** (*randomstream*) – randomstream to be used if omitted, random will be used if used as random.Random(12299) it assigns a new stream with the specified seed

mean()

Returns **Mean of the distribution**

Return type float

print_info(*as_str=False*, *file=None*)

prints information about the distribution

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

sample()

Returns **Sample of the distribution**

Return type float

class salabim.**Bounded**(*dis*, *lowerbound=None*, *upperbound=None*, *fail_value=None*, *number_of_retries=None*, *include_lowerbound=None*, *include_upperbound=None*, *time_unit=None*, *env=None*)

Parameters

- **dis** (*distribution*) – distribution to be bounded
- **lowerbound** (*float*) – sample values < lowerbound will be rejected (at most 100 retries) if omitted, no lowerbound check
- **upperbound** (*float*) – sample values > upperbound will be rejected (at most 100 retries) if omitted, no upperbound check
- **fail_value** (*float*) – value to be used if. after number_of_tries retries, sample is still not within bounds default: lowerbound, if specified, otherwise upperbound
- **number_of_tries** (*int*) – number of tries before fail_value is returned default: 100
- **include_lowerbound** (*bool*) – if True (default), the lowerbound may be included. if False, the lowerbound will be excluded.
- **include_upperbound** (*bool*) – if True (default), the upperbound may be included. if False, the upperbound will be excluded.

- **time_unit** (*str*) – specifies the time unit of the lowerbound or upperbound|n| must be one of "years", "weeks", "days", "hours", "minutes", "seconds", "milliseconds", "microseconds" default : no conversion

Note: If, after number_of_tries retries, the sampled value is still not within the given bounds, fail_value will be returned Samples that cannot be converted to float (only possible with Pdf and CumPdf) are assumed to be within the bounds.

mean()

Returns **Mean of the expression of bounded distribution** – unless no bounds are specified, returns nan

Return type float

print_info(*as_str=False*, *file=None*)
prints information about the expression of distribution(s)

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

class salabim.**Cdf**(*spec*, *time_unit=None*, *randomstream=None*, *env=None*)
Cumulative distribution function

Parameters

- **spec** (*list or tuple*) – list with x-values and corresponding cumulative density (x1,c1,x2,c2, . . . xn,cn) Requirements:

  x1<=x2<= . . . <=xn c1<=c2<=cn c1=0 cn>0 all cumulative densities are auto scaled according to cn, so no need to set cn to 1 or 100.
- **time_unit** (*str*) – specifies the time unit must be one of "years", "weeks", "days", "hours", "minutes", "seconds", "milliseconds", "microseconds" default : no conversion
- **randomstream** (*randomstream*) – if omitted, random will be used if used as random.Random(12299) it defines a new stream with the specified seed
- **env** (*Environment*) – environment where the distribution is defined if omitted, default_env will be used

mean()

Returns **Mean of the distribution**

Return type float

`print_info`(*as_str=False*, *file=None*)
prints information about the distribution

Parameters

- **`as_str`** (*`bool`*) – if False (default), print the info if True, return a string containing the info
- **`file`** (*`file`*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

`sample`()

Returns **Sample of the distribution**

Return type float

class salabim.**Constant**(*value*, *time_unit=None*, *randomstream=None*, *env=None*)
constant distribution

Parameters

- **`value`** (*`float`*) – value to be returned in sample
- **`time_unit`** (*`str`*) – specifies the time unit must be one of “years”, “weeks”, “days”, “hours”, “minutes”, “seconds”, “milliseconds”, “microseconds” default : no conversion
- **`randomstream`** (*`randomstream`*) – randomstream to be used if omitted, random will be used if used as random.Random(12299) it assigns a new stream with the specified seed Note that this is only for compatibility with other distributions
- **`env`** (`Environment`) – environment where the distribution is defined if omitted, default_env will be used

`mean`()

Returns **mean of the distribution (= the specified constant)**

Return type float

`print_info`(*as_str=False*, *file=None*)
prints information about the distribution

Parameters

- **`as_str`** (*`bool`*) – if False (default), print the info if True, return a string containing the info

- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

sample()

Returns **sample of the distribution (= the specified constant)**

Return type float

class salabim.**Distribution**(*spec*, *randomstream=None*, *time_unit=None*)

Generate a distribution from a string

Parameters

- **spec** (*str*) –
  - string containing a valid salabim distribution, where only the first letters are relevant and casing is not important. Note that Erlang, Cdf, CumPdf and Poisson require at least two letters (Er, Cd, Cu and Po)
  - string containing one float (c1), resulting in Constant(c1)
  - string containing two floats seperated by a comma (c1,c2), resulting in a Uniform(c1,c2)
  - string containing three floats, separated by commas (c1,c2,c3), resulting in a Triangular(c1,c2,c3)
- **time_unit** (*str*) – Supported time_units: “years”, “weeks”, “days”, “hours”, “minutes”, “seconds”, “milliseconds”, “microseconds” if spec has a time_unit as well, this parameter is ignored
- **randomstream** (*randomstream*) – if omitted, random will be used if used as random.Random(12299) it assigns a new stream with the specified seed

Note: The randomstream in the specifying string is ignored. It is possible to use expressions in the specification, as long these are valid within the context of the salabim module, which usually implies a global variable of the salabim package.

Examples

Uniform(13) ==> Uniform(13) Uni(12,15) ==> Uniform(12,15) UNIF(12,15) ==> Uniform(12,15) N(12,3) ==> Normal(12,3) Tri(10,20). ==> Triangular(10,20,15) 10. ==> Constant(10) 12,15 ==> Uniform(12,15) (12,15) ==> Uniform(12,15) Exp(a) ==> Exponential(100), provided sim.a=100 E(2) ==> Exponential(2) Er(2,3) ==> Erlang(2,3)

mean()

Returns **Mean of the distribution**

Return type float

print_info(*as_str=False*, *file=None*)
prints information about the distribution

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

sample()

Returns **Sample of the distribution**

Return type any (usually float)

class salabim.**Erlang**(*shape*, *rate=None*, *time_unit=None*, *scale=None*, *randomstream=None*, *env=None*)
erlang distribution

Parameters

- **shape** (*int*) – shape of the distribution (k) should be >0
- **rate** (*float*) – rate parameter (lambda) if omitted, the scale is used should be >0
- **time_unit** (*str*) – specifies the time unit must be one of "years", "weeks", "days", "hours", "minutes", "seconds", "milliseconds", "microseconds" default : no conversion
- **scale** (*float*) – scale of the distribution (mu) if omitted, the rate is used should be >0
- **randomstream** (*randomstream*) – randomstream to be used if omitted, random will be used if used as random.Random(12299) it assigns a new stream with the specified seed
- **env** (*Environment*) – environment where the distribution is defined if omitted, default_env will be used

Note: Either rate or scale has to be specified, not both.

mean()

Returns **Mean of the distribution**

Return type float

print_info(*as_str=False*, *file=None*)
prints information about the distribution

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

sample()

Returns **Sample of the distribution**

Return type float

class salabim.**Exponential**(*mean=None*, *time_unit=None*, *rate=None*, *randomstream=None*, *env=None*)
exponential distribution

Parameters

- **mean** (*float*) – mean of the distribtion (beta)|n| if omitted, the rate is used must be >0
- **time_unit** (*str*) – specifies the time unit must be one of "years", "weeks", "days", "hours", "minutes", "seconds", "milliseconds", "microseconds" default : no conversion
- **rate** (*float*) – rate of the distribution (lambda)|n| if omitted, the mean is used must be >0
- **randomstream** (*randomstream*) – randomstream to be used if omitted, random will be used if used as random.Random(12299) it assigns a new stream with the specified seed
- **env** (*Environment*) – environment where the distribution is defined if omitted, default_env will be used

Note: Either mean or rate has to be specified, not both

mean()

Returns **Mean of the distribution**

Return type float

print_info(*as_str=False*, *file=None*)
prints information about the distribution

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns info (if as_str is True)

Return type str

sample()

Returns Sample of the distribution

Return type float

class salabim.**External**(*dis*, **args*, ***kwargs*)
External distribution function

This distribution allows distributions from other modules, notably random, numpy.random and scipy.stats to be used as were they salabim distributions.

Parameters

- **dis** (*external distribution*) – either
  - random.xxx
  - numpy.random.xxx
  - scipy.stats.xxx
- ***args** (*any*) – positional arguments to be passed to the dis distribution
- ****kwargs** (*any*) – keyword arguments to be passed to the dis distribution
- **time_unit** (*str*) – specifies the time unit must be one of "years", "weeks", "days", "hours", "minutes", "seconds", "milliseconds", "microseconds" default : no conversion
- **env** (*Environment*) – environment where the distribution is defined if omitted, default_env will be used

mean()

Returns mean of the distribution – only available for scipy.stats distribution. Otherwise nan will be returned.

Return type float

`print_info`(*as_str=False*, *file=None*)
prints information about the distribution

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns info (if as_str is True)

Return type str

`sample`()

Returns Sample of the distribution via external distribution method

Return type any (usually float)

class salabim.**Gamma**(*shape*, *scale=None*, *time_unit=None*, *rate=None*, *randomstream=None*, *env=None*)
gamma distribution

Parameters

- **shape** (*float*) – shape of the distribution (k) should be >0
- **scale** (*float*) – scale of the distribution (teta) should be >0
- **time_unit** (*str*) – specifies the time unit must be one of "years", "weeks", "days", "hours", "minutes", "seconds", "milliseconds", "microseconds" default : no conversion
- **rate** (*float*) – rate of the distribution (beta) should be >0
- **randomstream** (*randomstream*) – randomstream to be used if omitted, random will be used if used as random.Random(12299) it assigns a new stream with the specified seed

env [Environment] environment where the distribution is defined if omitted, default_env will be used

Note: Either scale or rate has to be specified, not both.

`mean`()

Returns **Mean of the distribution**

Return type float

print_info(*as_str=False*, *file=None*)
prints information about the distribution

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

sample()

Returns **Sample of the distribution**

Return type float

class salabim.**Normal**(*mean*, *standard_deviation=None*, *time_unit=None*, *coefficient_of_variation=None*, *use_gauss=False*, *randomstream=None*, *env=None*)
normal distribution

Parameters

- **mean** (*float*) – mean of the distribution
- **standard_deviation** (*float*) – standard deviation of the distribution if omitted, coefficient_of_variation, is used to specify the variation if neither standard_devation nor coefficient_of_variation is given, 0 is used, thus effectively a contant distribution must be >=0
- **coefficient_of_variation** (*float*) – coefficient of variation of the distribution if omitted, standard_deviation is used to specify variation the resulting standard_deviation must be >=0
- **use_gauss** (*bool*) – if False (default), use the random.normalvariate method if True, use the random.gauss method the documentation for random states that the gauss method should be slightly faster, although that statement is doubtful.
- **time_unit** (*str*) – specifies the time unit must be one of "years", "weeks", "days", "hours", "minutes", "seconds", "milliseconds", "microseconds" default : no conversion
- **randomstream** (*randomstream*) – randomstream to be used if omitted, random will be used if used as random.Random(12299) it assigns a new stream with the specified seed
- **env** (*Environment*) – environment where the distribution is defined if omitted, default_env will be used

mean()

Returns **Mean of the distribution**

Return type float

print_info(*as_str=False*, *file=None*)
prints information about the distribution

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

sample()

Returns **Sample of the distribution**

Return type float

class salabim.**Pdf**(*spec*, *probabilities=None*, *time_unit=None*, *randomstream=None*, *env=None*)
Probability distribution function

Parameters

- **spec** (*list, tuple or dict*) – either
  - if no probabilities specified: list/tuple with x-values and corresponding probability dict where the keys are re x-values and the values are probabilities (x0, p0, x1, p1, . . . xn,pn)
  - if probabilities is specified: list with x-values
- **probabilities** (*iterable or float*) – if omitted, spec contains the probabilities the iterable (p0, p1, . . . pn) contains the probabilities of the corresponding x-values from spec. alternatively, if a float is given (e.g. 1), all x-values have equal probability. The value is not important.
- **time_unit** (*str*) – specifies the time unit must be one of "years", "weeks", "days", "hours", "minutes", "seconds", "milliseconds", "microseconds" default : no conversion
- **randomstream** (*randomstream*) – if omitted, random will be used if used as random.Random(12299) it assigns a new stream with the specified seed
- **env** (*Environment*) – environment where the distribution is defined if omitted, default_env will be used

Note: p0+p1=…+pn>0 all densities are auto scaled according to the sum of p0 to pn, so no need to have p0 to pn add up to 1 or 100. The x-values can be any type. If it is a salabim distribution, not the distribution, but a sample will be returned when calling sample.

mean()

Returns **mean of the distribution** – if the mean can't be calculated (if not all x-values are scalars or distributions), nan will be returned.

Return type float

print_info(*as_str=False*, *file=None*)

prints information about the distribution

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

sample(*n=None*)

Parameters **n** (*number of samples : int*) – if not specified, specifies just return one sample, as usual if specified, return a list of n sampled values from the distribution without replacement. This requires that all probabilities are equal. If n > number of values in the Pdf distribution, n is assumed to be the number of values in the distribution. If a sampled value is a distribution, a sample from that distribution will be returned.

Returns **Sample of the distribution** – In case n is specified, returns a list of n values

Return type any (usually float) or list

class salabim.**Poisson**(*mean*, *randomstream=None*)

Poisson distribution

Parameters

- **mean** (*float*) – mean (lambda) of the distribution
- **randomstream** (*randomstream*) – randomstream to be used if omitted, random will be used if used as random.Random(12299) it assigns a new stream with the specified seed

Note: The run time of this function increases when mean (lambda) increases. It is not recommended to use mean (lambda) > 100

mean()

Returns **Mean of the distribution**

Return type float

print_info(*as_str=False*, *file=None*)
prints information about the distribution

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

sample()

Returns **Sample of the distribution**

Return type int

class salabim.**Triangular**(*low*, *high=None*, *mode=None*, *time_unit=None*, *randomstream=None*, *env=None*)
triangular distribution

Parameters

- **low** (*float*) – lowerbound of the distribution
- **high** (*float*) – upperbound of the distribution if omitted, low will be used, thus effectively a constant distribution high must be >= low
- **mode** (*float*) – mode of the distribution if omitted, the average of low and high will be used, thus a symmetric triangular distribution mode must be between low and high
- **time_unit** (*str*) – specifies the time unit must be one of "years", "weeks", "days", "hours", "minutes", "seconds", "milliseconds", "microseconds" default : no conversion
- **randomstream** (*randomstream*) – randomstream to be used if omitted, random will be used if used as random.Random(12299) it assigns a new stream with the specified seed

- **env** (`Environment`) – environment where the distribution is defined if omitted, default_env will be used

mean()

Returns **Mean of the distribution**

Return type float

print_info(*as_str=False*, *file=None*)
prints information about the distribution

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

sample()

Returns **Sample of the distribtion**

Return type float

class salabim.**Uniform**(*lowerbound*, *upperbound=None*, *time_unit=None*, *randomstream=None*, *env=None*)
uniform distribution

Parameters

- **lowerbound** (*float*) – lowerbound of the distribution
- **upperbound** (*float*) – upperbound of the distribution if omitted, lowerbound will be used must be >= lowerbound
- **time_unit** (*str*) – specifies the time unit must be one of "years", "weeks", "days", "hours", "minutes", "seconds", "milliseconds", "microseconds" default : no conversion
- **randomstream** (*randomstream*) – randomstream to be used if omitted, random will be used if used as random.Random(12299) it assigns a new stream with the specified seed
- **env** (`Environment`) – environment where the distribution is defined if omitted, default_env will be used

mean()

Returns **Mean of the distribution**

Return type float

print_info(*as_str=False*, *file=None*)
prints information about the distribution

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns info (if as_str is True)

Return type str

sample()

Returns Sample of the distribution

Return type float

class salabim.**Weibull**(*scale*, *shape*, *time_unit=None*, *randomstream=None*, *env=None*)
weibull distribution

Parameters

- **scale** (*float*) – scale of the distribution (alpha or k)
- **shape** (*float*) – shape of the distribution (beta or lambda)|n| should be >0
- **time_unit** (*str*) – specifies the time unit must be one of "years", "weeks", "days", "hours", "minutes", "seconds", "milliseconds", "microseconds" default : no conversion
- **randomstream** (*randomstream*) – randomstream to be used if omitted, random will be used if used as random.Random(12299) it assigns a new stream with the specified seed
- **env** (Environment) – environment where the distribution is defined if omitted, default_env will be used

mean()

Returns Mean of the distribution

Return type float

print_info(*as_str=False*, *file=None*)
prints information about the distribution

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

sample()

Returns **Sample of the distribution**

Return type float

14.6 Environment

class salabim.**Environment**(*trace=False*, *random_seed=None*, *set_numpy_random_seed=True*, *time_unit='n/a'*, *datetime0=False*, *name=None*, *print_trace_header=True*, *isdefault_env=True*, *retina=False*, *do_reset=None*, *blind_animation=False*, **args*, ***kwargs*)

environment object

Parameters

- **trace** (*bool or file handle*) – defines whether to trace or not if this a file handle (open for write), the trace output will be sent to this file. if omitted, False
- **random_seed** (*hashable object, usually int*) – the seed for random, equivalent to random.seed() if "*", a purely random value (based on the current time) will be used (not reproducable) if the null string, no action on random is taken if None (the default), 1234567 will be used.
- **time_unit** (*str*) – Supported time_units: "years", "weeks", "days", "hours", "minutes", "seconds", "milliseconds", "microseconds", "n/a" default: "n/a"
- **datetime0** (*bool or datetime.datetime*) – display time and durations as datetime.datetime/datetime.timedelta if falsy (default), disabled if True, the t=0 will correspond to 1 January 1970 if no time_unit is specified, but datetime0 is not falsy, time_unit will be set to seconds
- **name** (*str*) – name of the environment if the name ends with a period (.), auto serializing will be applied if the name end with a comma, auto serializing starting at 1 will be applied if omitted, the name will be derived from the class (lowercased) or "default environment" if isdefault_env is True.
- **print_trace_header** (*bool*) – if True (default) print a (two line) header line as a legend if False, do not print a header note that the header is only printed if trace=True
- **isdefault_env** (*bool*) – if True (default), this environment becomes the default environment if False, this environment will not be the default environment if omitted, this environment becomes the default environment

- **set_numpy_random_seed** (*bool*) – if True (default), numpy.random.seed() will be called with the given seed. This is particularly useful when using External distributions. If numpy is not installed, this parameter is ignored if False, numpy.random.seed is not called.
- **do_reset** (*bool*) – if True, reset the simulation environment if False, do not reset the simulation environment if None (default), reset the simulation environment when run under Pythonista, otherwise no reset
- **blind_animation** (*bool*) – if False (default), animation will be performed as expected if True, animations will run silently. This is useful to make videos when tkinter is not installed (installable). This is particularly useful when running a simulation on a server. Note that this will show a slight performance increase, when creating videos.

Note: The trace may be switched on/off later with trace The seed may be later set with random_seed() Initially, the random stream will be seeded with the value 1234567. If required to be purely, not reproducable, values, use random_seed="*".

an_clocktext()
function to initialize the system clocktext called by run(), if animation is True. may be overridden to change the standard behaviour.

an_menu_buttons()
function to initialize the menu buttons may be overridden to change the standard behaviour.

an_modelname()
function to show the modelname may be overridden to change the standard behaviour.

an_synced_buttons()
function to initialize the synced buttons may be overridden to change the standard behaviour.

an_unsynced_buttons()
function to initialize the unsynced buttons may be overridden to change the standard behaviour.

animate(*value=None*)
animate indicator

Parameters **value** (*bool*) – new animate indicator if '?', animation will be set, if possible if not specified, no change

Returns **animate status**

Return type bool

Note: When the run is not issued, no action will be taken.

animate3d(*value=None*)
animate3d indicator

Parameters **value** (*bool*) – new animate3d indicator if '?', 3D-animation will be set, if possible if not specified, no change

Returns **animate3d status**

Return type bool

Note: When the animate is not issued, no action will be taken.

animate_debug(*value=None*)
Animate debug

Parameters **value** (*bool*) – animate_debug default: no change initially: False

Returns **animate_debug**

Return type bool

animation_parameters(*animate=None*, *synced=None*, *speed=None*, *width=None*, *height=None*, *title=None*, *show_menu_buttons=None*, *x0=None*, *y0=None*, *x1=None*, *background_color=None*, *foreground_color=None*, *background3d_color=None*, *fps=None*, *modelname=None*, *use_toplevel=None*, *show_fps=None*, *show_time=None*, *maximum_number_of_bitmaps=None*, *video=None*, *video_repeat=None*, *video_pingpong=None*, *audio=None*, *audio_speed=None*, *animate_debug=None*, *animate3d=None*, *width3d=None*, *height3d=None*, *video_width=None*, *video_height=None*, *video_mode=None*, *position=None*, *position3d=None*, *visible=None*)
set animation parameters

Parameters

- **animate** (*bool*) – animate indicator new animate indicator if '?', animation will be set, possible if not specified, no change
- **animate3d** (*bool*) – animate3d indicator new animate3d indicator if '?', 3D-animation will be set, possible if not specified, no change
- **synced** (*bool*) – specifies whether animation is synced if omitted, no change. At init of the environment synced will be set to True
- **speed** (*float*) – speed specifies how much faster or slower than real time the animation will run. e.g. if 2, 2 simulation time units will be displayed per second.
- **width** (*int*) – width of the animation in screen coordinates if omitted, no change. At init of the environment, the width will be set to 1024 for non Pythonista and the current screen width for Pythonista.
- **height** (*int*) – height of the animation in screen coordinates if omitted, no change. At init of the environment, the height will be set to 768 for non Pythonista and the current screen height for Pythonista.

- **position** (*tuple(x,y)*) – position of the animation window if omitted, no change. At init of the environment, the position will be set to (0, 0) no effect for Pythonista
- **width3d** (*int*) – width of the 3d animation in screen coordinates if omitted, no change. At init of the environment, the 3d width will be set to 1024.
- **height3d** (*int*) – height of the 3d animation in screen coordinates if omitted, no change. At init of the environment, the 3d height will be set to 768.
- **position3d** (*tuple(x,y)*) – position of the 3d animation window At init of the environment, the position will be set to (0, 0) This has to be set before the 3d animation starts as the window can only be postioned at initialization
- **title** (*str*) – title of the canvas window if omitted, no change. At init of the environment, the title will be set to salabim. if "", the title will be suppressed.
- **x0** (*float*) – user x-coordinate of the lower left corner if omitted, no change. At init of the environment, x0 will be set to 0.
- **y0** (*float*) – user y_coordinate of the lower left corner if omitted, no change. At init of the environment, y0 will be set to 0.
- **x1** (*float*) – user x-coordinate of the lower right corner if omitted, no change. At init of the environment, x1 will be set to 1024 for non Pythonista and the current screen width for Pythonista.
- **background_color** (*colorspec*) – color of the background if omitted, no change. At init of the environment, this will be set to white.
- **foreground_color** (*colorspec*) – color of foreground (texts) if omitted and background_color is specified, either white of black will be used, in order to get a good contrast with the background color. if omitted and background_color is also omitted, no change. At init of the environment, this will be set to black.
- **background3d_color** (*colorspec*) – color of the 3d background if omitted, no change. At init of the environment, this will be set to black.
- **fps** (*float*) – number of frames per second
- **modelname** (*str*) – name of model to be shown in upper left corner, along with text "a salabim model" if omitted, no change. At init of the environment, this will be set to the null string, which implies suppression of this feature.
- **use_toplevel** (*bool*) – if salabim animation is used in parallel with other modules using tkinter, it might be necessary to initialize the root with tkinter.TopLevel(). In that case, set this parameter to True. if False (default), the root will be initialized with tkinter.Tk()
- **show_fps** (*bool*) – if True, show the number of frames per second if False, do not show the number of frames per second (default)
- **show_time** (*bool*) – if True, show the time (default) if False, do not show the time
- **show_menu_buttons** (*bool*) – if True, show the menu buttons (default) if False, do not show the menu buttons
- **maximum_number_of_bitmaps** (*int*) – maximum number of tkinter bitmaps (default 4000)

- **video** (*str*) – if video is not omitted, a video with the name video will be created. Normally, use .mp4 as extension. If the extension is .gif or .png an animated gif / png file will be written, unless there is a * in the filename If the extension is .gif, .jpg, .png, .bmp, .ico or .tiff and one * appears in the filename, individual frames will be written with a six digit sequence at the place of the asteriks in the file name. If the video extension is not .gif, .jpg, .png, .bmp, .ico or .tiff, a codec may be added by appending a plus sign and the four letter code name, like "myvideo.avi+DIVX". If no codec is given, MJPG will be used for .avi files, otherwise .mp4v Under PyDroid only .avi files are supported.
- **video_repeat** (*int*) – number of times animated gif or png should be repeated 0 means inifinite at init of the environment video_repeat is 1 this only applies to gif and png files production.
- **video_pingpong** (*bool*) – if True, all frames will be added reversed at the end of the video (useful for smooth loops) at init of the environment video_pingpong is False this only applies to gif and png files production.
- **audio** (*str*) – name of file to be played (mp3 or wav files) if the none string, the audio will be stopped default: no change for more information, see Environment.audio()
- **visible** (*bool*) – if True (start condition), the animation window will be visible if False, the animation window will be hidden ('withdrawn')

Note: The y-coordinate of the upper right corner is determined automatically in such a way that the x and y scaling are the same.

animation_post_tick (*t*)
called just after the animation object loop. Default behaviour: just return

Parameters **t** (*float*) – Current (animation) time.

animation_pre_tick (*t*)
called just before the animation object loop. Default behaviour: just return

Parameters **t** (*float*) – Current (animation) time.

audio (*filename*)
Play audio during animation

Parameters **filename** (*str*) – name of file to be played (mp3 or wav files) if "", the audio will be stopped optionaly, a start time in seconds may be given by appending the filename a > followed by the start time, like 'mytune.mp3>12.5' if not specified (None), no change

Returns **filename being played ("" if nothing is being played)**

Return type str

Note: Only supported on Windows and Pythonista platforms. On other platforms, no effect. Variable bit rate mp3 files may be played incorrectly on Windows platforms. Try and use fixed bit rates (e.g. 128 or 320 kbps)

audio_speed(*value=None*)
Play audio during animation

Parameters **value** (*float*) – animation speed at which the audio should be played default: no change initially: 1

Returns **speed being played**

Return type int

background3d_color(*value=None*)
background3d_color of the animation

Parameters **value** (*colorspec*) – new background_color if not specified, no change

Returns **background3d_color of animation**

Return type colorspec

background_color(*value=None*)
background_color of the animation

Parameters **value** (*colorspec*) – new background_color if not specified, no change

Returns **background_color of animation**

Return type colorspec

base_name()
returns the base name of the environment (the name used at initialization)

beep()
Beeps

Works only on Windows and iOS (Pythonista). For other platforms this is just a dummy method.

camera_auto_print(*value=None*)
queries or set camera_auto_print

Parameters **value** (*boolean*) – if None (default), no action if True, camera_print will be called on each camera control keypress if False, no automatic camera_print

Returns **Current status**

Return type bool

Note: The camera_auto_print functionality is useful to get the spec for camera_move()

camera_move (*spec=", lag=1, offset=0, enabled=True*)
Moves the camera according to the given spec, which is normally a collection of camera_print outputs.

Parameters

- **spec** (*str*) – output normally obtained from camera_auto_print lines
- **lag** (*float*) – lag time (for smooth camera movements) (default: 1))
- **offset** (*float*) – the duration (can be negative) given is added to the times given in spec. Default: 0
- **enabled** (*bool*) – if True (default), move camera according to spec/lag if False, freeze camera movement

color_interp (*x, xp, fp*)
linear interpolation of a color

Parameters

- **x** (*float*) – target x-value
- **xp** (*list of float, tuples or lists*) – values on the x-axis
- **fp** (*list of colorspecs*) – values on the y-axis should be same length as xp

Returns interpolated color value

Return type tuple

Notes

If x < xp[0], fp[0] will be returned If x > xp[-1], fp[-1] will be returned

colorinterpolate (*t, t0, t1, v0, v1*)
does linear interpolation of colorspecs

Parameters

- **t** (*float*) – value to be interpolated from
- **t0** (*float*) – f(t0)=v0
- **t1** (*float*) – f(t1)=v1

- **`v0`** (*`colorspec`*) – f(t0)=v0
- **`v1`** (*`colorspec`*) – f(t1)=v1

Returns **linear interpolation between v0 and v1 based on t between t0 and t**

Return type colorspec

Note: Note that no extrapolation is done, so if t<t0 ==> v0 and t>t1 ==> v1 This function is heavily used during animation

`colorspec_to_tuple`(*colorspec*)
translates a colorspec to a tuple

Parameters **`colorspec`** (*`tuple, list or str`*) – `#rrggbb` ==> alpha = 255 (rr, gg, bb in hex) `#rrggbbaa` ==> alpha = aa (rr, gg, bb, aa in hex) `colorname` ==> alpha = 255 `(colorname, alpha)` `(r, g, b)` ==> alpha = 255 `(r, g, b, alpha)` `"fg"` ==> foreground_color `"bg"` ==> background_color

Returns

Return type (r, g, b, a)

`current_component`()

Returns **the current_component**

Return type *Component*

`datetime0`(*datetime0=None*)
Gets and/or sets datetime0

Parameters **`datetime0`** (*`bool or datetime.datetime`*) – if omitted, nothing will be set if falsy, disabled if True, the t=0 will correspond to 1 January 1970 if no time_unit is specified, but datetime0 is not falsy, time_unit will be set to seconds

Returns **current value of datetime0**

Return type bool or datetime.datetime

`datetime_to_t`(*datetime*)

Parameters **`datetime`** (*`datetime.datetime`*) –

Returns **datetime translated to simulation time in the current time_unit**

Return type float

Raises `ValueError` – if datetime0 is False

days(*t*)
convert the given time in days to the current time unit

Parameters **t** (*float or distribution*) – time in days if distribution, the distribution is sampled

Returns **time in days, converted to the current time_unit**

Return type float

duration_to_str(*duration*)

Parameters **duration** (*float*) – duration to be converted to string in trace

Returns **duration in required format** – default: f"{duration:.3f}" if datetime0 is False or duration in the format "hh:mm:dd" or "d hh:mm:ss"

Return type str

Note: May be overrridden.

duration_to_timedelta(*duration*)

Parameters **duration** (*float*) –

Returns **timedelta corresponding to duration**

Return type datetime.timedelta

Raises `ValueError` – if time unit is not set

foreground_color(*value=None*)
foreground_color of the animation

Parameters **value** (*colorspec*) – new foreground_color if not specified, no change

Returns **foreground_color of animation**

Return type colorspec

fps(*value=None*)

Parameters **value** (*int*) – new fps if not specified, no change

Returns **fps**

Return type bool

`get_time_unit`()
gets time unit

Returns Current time unit dimension (default "n/a")

Return type str

`height`(*value=None*)
height of the animation in screen coordinates

Parameters **`value`** (*`int`*) – new height if not specified, no change

Returns height of animation

Return type int

`height3d`(*value=None*)
height of the 3d animation in screen coordinates

Parameters **`value`** (*`int`*) – new 3d height if not specified, no change

Returns height of 3d animation

Return type int

`hours`(*t*)
convert the given time in hours to the current time unit

Parameters **`t`** (*`float or distribution`*) – time in hours if distribution, the distribution is sampled

Returns time in hours, converted to the current time_unit

Return type float

`insert_frame`(*image*, *number_of_frames=1*)
Insert image as frame(s) into video

Parameters

- **`image`** (*`Pillow image, str or Path object`*) – Image to be inserted
- **`nuumber_of_frames`** (*`int`*) – Number of 1/30 second long frames to be inserted

`is_dark`(*colorspec*)

Parameters **`colorspec`** (*`colorspec`*) – color to check

Returns True, if the colorspec is dark (rather black than white) False, if the colorspec is light (rather white than black if colorspec has alpha=0 (total transparent), the background_color will be tested

Return type bool

is_videoing()

video recording status

Returns **video recording status** – True, if video is being recorded False, otherwise

Return type bool

main()

Returns **the main component**

Return type *Component*

maximum_number_of_bitmaps(*value=None*)

maximum number of bitmaps (applies to animation with tkinter only)

Parameters **value** (*int*) – new maximum_number_of_bitmaps if not specified, no change

Returns **maximum number of bitmaps**

Return type int

microseconds(*t*)

convert the given time in microseconds to the current time unit

Parameters **t** (*float or distribution*) – time in microseconds if distribution, the distribution is sampled

Returns **time in microseconds, converted to the current time_unit**

Return type float

milliseconds(*t*)

convert the given time in milliseconds to the current time unit

Parameters **t** (*float or distribution*) – time in milliseconds if distribution, the distribution is sampled

Returns **time in milliseconds, converted to the current time_unit**

Return type float

minutes(*t*)

convert the given time in minutes to the current time unit

Parameters **t** (*float or distribution*) – time in minutes if distribution, the distribution is sampled

Returns **time in minutes, converted to the current time_unit**

Return type float

modelname(*value=None*)

Parameters **value** (*str*) – new modelname if not specified, no change

Returns **modelname**

Return type str

Note: If modelname is the null string, nothing will be displayed.

name(*value=None*)

Parameters **value** (*str*) – new name of the environment if omitted, no change

Returns **Name of the environment**

Return type str

Note: base_name and sequence_number are not affected if the name is changed

now()

Returns **the current simulation time**

Return type float

peek()

returns the time of the next component to become current if there are no more events, peek will return inf Only for advance use with animation / GUI event loops

position(*value=None*)

position of the animation window

Parameters **value** (*tuple (x, y)*) – new position if not specified, no change

Returns **position of animation window**

Return type tuple (x,y)

position3d(*value=None*)

position of the 3d animation window

Parameters **value** (*tuple (x, y)*) – new position if not specified, no change

Returns **position of th 3d animation window**

Return type tuple (x,y)

Note: This must be given before the 3d animation is started.

---

print_info(*as_str=False*, *file=None*)

prints information about the environment

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

print_trace(*s1=”*, *s2=”*, *s3=”*, *s4=”*, *s0=None*, *_optional=False*)

prints a trace line

Parameters

- **s1** (*str*) – part 1 (usually formatted now), padded to 10 characters
- **s2** (*str*) – part 2 (usually only used for the compoent that gets current), padded to 20 characters
- **s3** (*str*) – part 3, padded to 35 characters
- **s4** (*str*) – part 4
- **s0** (*str*) – part 0. if omitted, the line number from where the call was given will be used at the start of the line. Otherwise s0, left padded to 7 characters will be used at the start of the line.
- **_optional** (*bool*) – for internal use only. Do not set this flag!

Note: if self.trace is False, nothing is printed if the current component’s suppress_trace is True, nothing is printed

---

print_trace_header()
print a (two line) header line as a legend also the legend for line numbers will be printed not that the header is only printed if trace=True

reset_now(*new_now=0*)
reset the current time

Parameters **new_now** (*float or distribution*) – now will be set to new_now default: 0 if distribution, the distribution is sampled

---

Note: Internally, salabim still works with the 'old' time. Only in the interface from and to the user program, a correction will be applied.

The registered time in monitors will be always is the 'old' time. This is only relevant when using the time value in Monitor.xt() or Monitor.tx().

---

run(*duration=None*, *till=None*, *priority=inf*, *urgent=False*, *cap_now=None*)
start execution of the simulation

Parameters

- **duration** (*float or distribution*) – schedule with a delay of duration if 0, now is used if distribution, the distribution is sampled
- **till** (*float or distribution*) – schedule time if omitted, inf is assumed. See also note below if distribution, the distribution is sampled
- **priority** (*float*) – priority default: inf if a component has the same time on the event list, main is sorted accoring to the priority. The default value of inf makes that all components will finish before the run is ended
- **urgent** (*bool*) – urgency indicator if False (default), main will be scheduled behind all other components scheduled with the same time and priority if True, main will be scheduled in front of all components scheduled for the same time and priority
- **cap_now** (*bool*) – indicator whether times (till, duration) in the past are allowed. If, so now() will be used. default: sys.default_cap_now(), usualy False

---

Note: if neither till nor duration is specified, the main component will be reactivated at the time there are no more events on the eventlist, i.e. possibly not at inf. if you want to run till inf (particularly when animating), issue run(sim.inf) only issue run() from the main level

---

scale()
scale of the animation, i.e. width / (x1 - x0)

Returns scale

Return type float

Note: It is not possible to set this value explicitely.

`screen_to_user_coordinates_size`(*screensize*)
converts a screen size to a value to be used with user coordinates

Parameters **`screensize`** (*`float`*) – screen size to be converted

Returns **value corresponding with screensize in user coordinates**

Return type float

`screen_to_user_coordinates_x`(*screenx*)
converts a screen x coordinate to a user x coordinate

Parameters **`screenx`** (*`float`*) – screen x coordinate to be converted

Returns **user x coordinate**

Return type float

`screen_to_user_coordinates_y`(*screeny*)
converts a screen x coordinate to a user x coordinate

Parameters **`screeny`** (*`float`*) – screen y coordinate to be converted

Returns **user y coordinate**

Return type float

`seconds`(*t*)
convert the given time in seconds to the current time unit

Parameters **`t`** (*`float or distribution`*) – time in seconds if distribution, the distribution is sampled

Returns **time in secoonds, converted to the current time_unit**

Return type float

`sequence_number`()

Returns **sequence_number of the environment** – (the sequence number at initialization) normally this will be the integer value of a serialized name, but also non serialized names (without a dot or a comma at the end) will be numbered)

Return type int

`setup`()
called immediately after initialization of an environment.

by default this is a dummy method, but it can be overridden.

only keyword arguments are passed

`show_camera_position`(*over3d=None*)
show camera position on the tkinter window or over3d window

The 7 camera settings will be shown in the top left corner.

Parameters **`over3d`** (*`bool`*) – if False (default), present on 2D screen if True, present on 3D overlay

`show_fps`(*value=None*)

Parameters **`value`** (*`bool`*) – new show_fps if not specified, no change

Returns **show_fps**

Return type bool

`show_menu_buttons`(*value=None*)
controls menu buttons

Parameters **`value`** (*`bool`*) – if True, menu buttons are shown if False, menu buttons are hidden if not specified, no change

Returns **show menu button status**

Return type bool

`show_time`(*value=None*)

Parameters **`value`** (*`bool`*) – new show_time if not specified, no change

Returns **show_time**

Return type bool

`snapshot`(*filename*, *video_mode='2d'*)
Takes a snapshot of the current animated frame (at time = now()) and saves it to a file

Parameters

- **`filename`** (*`str`*) – file to save the current animated frame to. The following formats are accepted: .png, .jpg, .bmp, .ico, .gif and .tiff. Other formats are not possible. Note that, apart from .JPG files. the background may be semi transparent by setting the alpha value to something else than 255.

- **video_mode** (*str*) – specifies what to save if "2d" (default), the tkinter window will be saved if "3d", the OpenGL window will be saved (provided animate3d is True) if "screen" the complete screen will be saved (no need to be in animate mode)|n| no scaling will be applied.

speed(*value=None*)

Parameters **value** (*float*) – new speed if not specified, no change

Returns **speed**

Return type float

step()

executes the next step of the future event list

for advanced use with animation / GUI loops

suppress_trace_linenumbers(*value=None*)

indicates whether line numbers should be suppressed (False by default)

Parameters **value** (*bool*) – new suppress_trace_linenumbers status if omitted, no change

Returns **suppress_trace_linenumbers status**

Return type bool

Note: By default, suppress_trace_linenumbers is False, meaning that line numbers are shown in the trace. In order to improve performance, line numbers can be suppressed.

suppress_trace_standby(*value=None*)

suppress_trace_standby status

Parameters **value** (*bool*) – new suppress_trace_standby status if omitted, no change

Returns **suppress trace status**

Return type bool

Note: By default, suppress_trace_standby is True, meaning that standby components are (apart from when they become non standby) suppressed from the trace. If you set suppress_trace_standby to False, standby components are fully traced.

synced(*value=None*)

Parameters **`value`** (*`bool`*) – new synced if not specified, no change

Returns **synced**

Return type bool

`t`()

Returns **the current simulation animation time**

Return type float

`t_to_datetime`(*t*)

Parameters **`t`** (*`float`*) – time to convert

Returns **t (in the current time unit) translated to the corresponding datetime**

Return type float

Raises `ValueError` – if datetime0 is False

`time_to_str`(*t*)

Parameters **`t`** (*`float`*) – time to be converted to string in trace and animation

Returns **t in required format** – default: f"{t:10.3f}" if datetime0 is False or date in the format "Day YYYY-MM-DD hh:mm:dd" otherwise

Return type str

Note: May be overrridden. Make sure that the method always returns the same length!

`timedelta_to_duration`(*timedelta*)

Parameters **`timedelta`** (*`datetime.timedelta`*) –

Returns **timedelta translated to simulation duration in the current time_unit**

Return type float

Raises `ValueError` – if datetime0 is False

`title`(*value=None*)
title of the canvas window

Parameters **`value`** (*`str`*) – new title if "", the title will be suppressed if not specified, no change

Returns **title of canvas window**

Return type str

Note: No effect for Pythonista

to_days (*t*)
convert time t to days

Parameters **t** (*float or distribution*) – time to be converted if distribution, the distribution is sampled

Returns **Time t converted to days**

Return type float

to_hours (*t*)
convert time t to hours

Parameters **t** (*float or distribution*) – time to be converted if distribution, the distribution is sampled

Returns **Time t converted to hours**

Return type float

to_microseconds (*t*)
convert time t to microseconds

Parameters **t** (*float or distribution*) – time to be converted if distribution, the distribution is sampled

Returns **Time t converted to microseconds**

Return type float

to_milliseconds (*t*)
convert time t to milliseconds

Parameters **t** (*float or distribution*) – time to be converted if distribution, the distribution is sampled

Returns **Time t converted to milliseconds**

Return type float

to_minutes (*t*)
convert time t to minutes

Parameters **t** (*float or distribution*) – time to be converted if distribution, the distribution is sampled

Returns **Time t converted to minutes**

Return type float

to_seconds (*t*)
convert time t to seconds

Parameters **t** (*float or distribution*) – time to be converted if distribution, the distribution is sampled

Returns **Time t converted to seconds**

Return type float

to_time_unit (*time_unit*, *t*)
convert time t to the time_unit specified

Parameters

- **time_unit** (*str*) – Supported time_units: "years", "weeks", "days", "hours", "minutes", "seconds", "milliseconds", "microseconds"
- **t** (*float or distribution*) – time to be converted if distribution, the distribution is sampled

Returns **Time t converted to the time_unit specified**

Return type float

to_weeks (*t*)
convert time t to weeks

Parameters **t** (*float or distribution*) – time to be converted if distribution, the distribution is sampled

Returns **Time t converted to weeks**

Return type float

to_years (*t*)
convert time t to years

Parameters **t** (*float or distribution*) – time to be converted if distribution, the distribution is sampled

Returns **Time t converted to years**

Return type float

trace (*value=None*)
trace status

Parameters **value** (*bool of file handle*) – new trace status defines whether to trace or not if this a file handle (open for write), the trace output will be sent to this file. if omitted, no change

Returns **trace status**

Return type bool or file handle

Note: If you want to test the status, always include parentheses, like

```
if env.trace():
```

user_to_screen_coordinates_size(*usersize*)
converts a user size to a value to be used with screen coordinates

Parameters **usersize** (*float*) – user size to be converted

Returns **value corresponding with usersize in screen coordinates**

Return type float

user_to_screen_coordinates_x(*userx*)
converts a user x coordinate to a screen x coordinate

Parameters **userx** (*float*) – user x coordinate to be converted

Returns **screen x coordinate**

Return type float

user_to_screen_coordinates_y(*usery*)
converts a user x coordinate to a screen x coordinate

Parameters **usery** (*float*) – user y coordinate to be converted

Returns **screen y coordinate**

Return type float

video(*value*)
video name

Parameters **value** (*str, list or tuple*) – new video name for explanation see animation_parameters()

Note: If video is the null string ro None, the video (if any) will be closed. The call can be also used as a context manager, which automatically opens and closes a file. E.g.

```
with video("test.mp4"):
    env.run(100)
```

video_close()
closes the current animation video recording, if any.

video_height(*value=None*)
height of the video animation in screen coordinates

Parameters **value** (*int*) – new width if not specified, no change

Returns **height of video animation**

Return type int

video_mode(*value=None*)

Parameters **value** (*int*) – new video mode ("2d", "3d" or "screen") if not specified, no change

Returns **video_mode**

Return type int

video_pingpong(*value=None*)
video pingpong

Parameters **value** (*bool*) – new video pingpong if not specified, no change

Returns **video pingpong**

Return type bool

Note: Applies only to gif animation.

video_repeat(*value=None*)
video repeat

Parameters **value** (*int*) – new video repeat if not specified, no change

Returns **video repeat**

Return type int

Note: Applies only to gif animation.

`video_width`(*value=None*)
width of the video animation in screen coordinates

Parameters **`value`** (`int`) – new width if not specified, no change

Returns **width of video animation**

Return type int

`visible`(*value=None*)
controls visibility of the animation window

Parameters **`value`** (`bool`) – if True, the animation window will be visible if False, the animation window will be hidden ('withdrawn') if None (default), no change

Returns **current visibility**

Return type bool

`weeks`(*t*)
convert the given time in weeks to the current time unit

Parameters **`t`** (`float or distribution`) – time in weeks if distribution, the distribution is sampled

Returns **time in weeks, converted to the current time_unit**

Return type float

`width`(*value=None*)
width of the animation in screen coordinates

Parameters **`value`** (`int`) – new width if not specified, no change

Returns **width of animation**

Return type int

`width3d`(*value=None*)
width of the 3d animation in screen coordinates

Parameters **`value`** (*`int`*) – new 3d width if not specified, no change

Returns **width of 3d animation**

Return type int

`x0`(*value=None*)

x coordinate of lower left corner of animation

Parameters **`value`** (*`float`*) – new x coordinate

Returns **x coordinate of lower left corner of animation**

Return type float

`x1`(*value=None*)

x coordinate of upper right corner of animation : float

Parameters **`value`** (*`float`*) – new x coordinate if not specified, no change

Returns **x coordinate of upper right corner of animation**

Return type float

`y0`(*value=None*)

y coordinate of lower left corner of animation

Parameters **`value`** (*`float`*) – new y coordinate if not specified, no change

Returns **y coordinate of lower left corner of animation**

Return type float

`y1`()

y coordinate of upper right corner of animation

Returns **y coordinate of upper right corner of animation**

Return type float

Note: It is not possible to set this value explicitely.

`years`(*t*)

convert the given time in years to the current time unit

Parameters **t** (*float or distribution*) – time in years if distribution, the distribution is sampled

Returns **time in years, converted to the current time_unit**

Return type float

14.7 ItemFile

class salabim.**ItemFile**(*filename*)

define an item file to be used with read_item, read_item_int, read_item_float and read_item_bool

Parameters **filename** (*str*) – file to be used for subsequent read_item, read_item_int, read_item_float and read_item_bool calls or content to be interpreted used in subsequent read_item calls. The content should have at least one linefeed character and will be usually triple quoted.

Note: It is advised to use ItemFile with a context manager, like

```
with sim.ItemFile("experiment0.txt") as f:
    run_length = f.read_item_float() |n|
    run_name = f.read_item() |n|
```

Alternatively, the file can be opened and closed explicitely, like

```
f = sim.ItemFile("experiment0.txt")
run_length = f.read_item_float()
run_name = f.read_item()
f.close()
```

Item files consist of individual items separated by whitespace (blank or tab)|n| If a blank or tab is required in an item, use single or double quotes All text following # on a line is ignored All texts on a line within curly brackets {} is ignored and considered white space. Curly braces cannot spawn multiple lines and cannot be nested.

Example

```
Item1
"Item 2"
    Item3 Item4 # comment
Item5 {five} Item6 {six}
'Double quote" in item'
"Single quote' in item"
True
```

read_item()
: read next item from the ItemFile

  if the end of file is reached, EOFError is raised

read_item_bool()
: read next item from the ItemFile as bool

  A value of False (not case sensitive) will return False A value of 0 will return False The null string will return False Any other value will return True

  if the end of file is reached, EOFError is raised

read_item_float()
: read next item from the ItemFile as float

  if the end of file is reached, EOFError is raised

read_item_int()
: read next field from the ItemFile as int.h

  if the end of file is reached, EOFError is raised

14.8 Monitor

class salabim.**Monitor**(*name=None*, *monitor=True*, *level=False*, *initial_tally=None*, *type=None*, *weight_legend=None*, *fill=None*, *stats_only=False*, *env=None*, **args*, ***kwargs*)

Monitor object

Parameters

- **name** (*str*) – name of the monitor if the name ends with a period (.), auto serializing will be applied if the name end with a comma, auto serializing starting at 1 will be applied if omitted, the name will be derived from the class it is defined in (lowercased)
- **monitor** (*bool*) – if True (default), monitoring will be on. if False, monitoring is disabled it is possible to control monitoring later, with the monitor method
- **level** (*bool*) – if False (default), individual values are tallied, optionally with weight if True, the tallied vslues are interpreted as levels
- **initial_tally** (*any, preferably int, float or translatable into int or float*) – initial value for the a level monitor it is important to set the value correctly. default: 0 not available for non level monitors
- **type** (*str*) –

  specifies how tallied values are to be stored

  - **"any" (default) stores values in a list. This allows** non numeric values. In calculations the values are forced to a numeric value (0 if not possible)
  - "bool" (True, False) Actually integer >= 0 <= 255 1 byte
  - "int8" integer >= -128 <= 127 1 byte
  - "uint8" integer >= 0 <= 255 1 byte
  - "int16" integer >= -32768 <= 32767 2 bytes
  - "uint16" integer >= 0 <= 65535 2 bytes
  - "int32" integer >= -2147483648<= 2147483647 4 bytes
  - "uint32" integer >= 0 <= 4294967295 4 bytes
  - "int64" integer >= -9223372036854775808 <= 9223372036854775807 8 bytes
  - "uint64" integer >= 0 <= 18446744073709551615 8 bytes
  - "float" float 8 bytes

- **weight_legend** (*str*) – used in print_statistics and print_histogram to indicate the dimension of weight or duration (for level monitors, e.g. minutes. Default: weight for non level monitors, duration for level monitors.
- **stats_only** (*bool*) – if True, only statistics will be collected (using less memory, but also less functionality) if False (default), full functionality
- **fill** (*list or tuple*) – can be used to fill the tallied values (all at time now). fill is only available for non level and not stats_only monitors.
- **env** (*Environment*) – environment where the monitor is defined if omitted, default_env will be used

animate(**args*, ***kwargs*)

animates the monitor in a panel

Parameters

- **linecolor** (*colorspec*) – color of the line or points (default foreground color)
- **linewidth** (*int*) – width of the line or points (default 1 for level, 3 for non level monitors)
- **fillcolor** (*colorspec*) – color of the panel (default transparent)
- **bordercolor** (*colorspec*) – color of the border (default foreground color)
- **borderlinewidth** (*int*) – width of the line around the panel (default 1)
- **nowcolor** (*colorspec*) – color of the line indicating now (default red)

- **titlecolor** (*colorspec*) – color of the title (default foreground color)
- **titlefont** (*font*) – font of the title (default null string)
- **titlefontsize** (*int*) – size of the font of the title (default 15)
- **title** (*str*) – title to be shown above panel default: name of the monitor
- **x** (*int*) – x-coordinate of panel, relative to xy_anchor, default 0
- **y** (*int*) – y-coordinate of panel, relative to xy_anchor. default 0
- **offsetx** (*float*) – offsets the x-coordinate of the panel (default 0)
- **offsety** (*float*) – offsets the y-coordinate of the panel (default 0)
- **angle** (*float*) – rotation angle in degrees, default 0
- **xy_anchor** (*str*) – specifies where x and y are relative to possible values are (default: sw): `nw n ne w c e sw s se`
- **vertical_offset** (*float*) –

  the vertical position of x within the panel is vertical_offset + x * vertical_scale (default 0)
- **vertical_scale** (*float*) – the vertical position of x within the panel is vertical_offset + x * vertical_scale (default 5)
- **horizontal_scale** (*float*) – the relative horizontal position of time t within the panel is on t * horizontal_scale, possibly shifted (default 1)|n|
- **width** (*int*) – width of the panel (default 200)
- **height** (*int*) – height of the panel (default 75)
- **vertical_map** (*function*) – when a y-value has to be plotted it will be translated by this function default: float when the function results in a TypeError or ValueError, the value 0 is assumed when y-values are non numeric, it is advised to provide an approriate map function, like: vertical_map = "unknown red green blue yellow".split().index
- **labels** (*iterable*) – labels to be shown on the vertical axis (default: empty tuple) the placement of the labels is controlled by the vertical_map method
- **label_color** (*colorspec*) – color of labels (default: foreground color)
- **label_font** (*font*) – font of the labels (default null string)
- **label_fontsize** (*int*) – size of the font of the labels (default 15)
- **label_anchor** (*str*) – specifies where the label coordinates (as returned by map_value) are relative to possible values are (default: e): `nw n ne w c e sw s se`

- **label_offsetx** (*float*) – offsets the x-coordinate of the label (default 0)
- **label_offsety** (*float*) – offsets the y-coordinate of the label (default 0)
- **label_linewidth** (*int*) – width of the label line (default 1)
- **label_linecolor** (*colorspec*) – color of the label lines (default foreground color)
- **layer** (*int*) – layer (default 0)
- **as_points** (*bool*) – allows to override the as_points setting of tallies, which is by default False for level monitors and True for non level monitors
- **parent** (Component) – component where this animation object belongs to (default None) if given, the animation object will be removed automatically when the parent component is no longer accessible
- **screen_coordinates** (*bool*) – use screen_coordinates normally, the scale parameters are use for positioning and scaling objects. if True, screen_coordinates will be used instead.
- **over3d** (*bool*) – if True, this object will be rendered to the OpenGL window if False (default), the normal 2D plane will be used.

Returns **reference to AnimateMonitor object**

Return type *AnimateMonitor*

Note: All measures are in screen coordinates

Note: It is recommended to use sim.AnimateMonitor instead

All measures are in screen coordinates

base_name()

Returns **base name of the monitor (the name used at initialization)**

Return type str

bin_duration(*lowerbound*, *upperbound*)

total duration of tallied values in range (lowerbound,upperbound]

Parameters

- **lowerbound** (*float*) – non inclusive lowerbound

- **upperbound** (*float*) – inclusive upperbound
- **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes

Returns **total duration of values >lowerbound and <=upperbound**

Return type int

Note: Not available for level monitors

bin_number_of_entries(*lowerbound*, *upperbound*, *ex0=False*)
count of the number of tallied values in range (lowerbound,upperbound]

Parameters

- **lowerbound** (*float*) – non inclusive lowerbound
- **upperbound** (*float*) – inclusive upperbound
- **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes

Returns **number of values >lowerbound and <=upperbound**

Return type int

Note: Not available for level monitors

bin_weight(*lowerbound*, *upperbound*)
total weight of tallied values in range (lowerbound,upperbound]

Parameters

- **lowerbound** (*float*) – non inclusive lowerbound
- **upperbound** (*float*) – inclusive upperbound
- **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes

Returns **total weight of values >lowerbound and <=upperbound**

Return type int

Note: Not available for level monitors

deregister(*registry*)
deregisters the monitor in the registry

Parameters **registry** (*list*) – list of registered objects

Returns **monitor (self)**

Return type *Monitor*

duration(*ex0=False*)
total duration

Parameters **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes

Returns **total duration**

Return type float

Note: Not available for non level monitors

duration_zero()
total duratiom of zero entries

Returns **total duration of zero entries**

Return type float

Note: Not available for non level monitors

freeze(*name=None*)
freezes this monitor (particularly useful for pickling)

Parameters **name** (*str*) – name of the frozen monitor default: name of this monitor + ".frozen"

Returns **frozen monitor**

Return type *Monitor*

Notes

The env attribute will become a partial copy of the original environment, with the name of the original environment, padded with '.copy.<serial number>'

get(*t=None*)
get the value of a level monitor

Parameters **t** (*float*) – time at which the value of the level is to be returned default: now

Returns

last tallied value – Instead of this method, the level monitor can also be called directly, like

level = sim.Monitor("level", level=True) … print(level()) print(level.get()) # identical

Return type any, usually float

Note: If the value is not available, self.off will be returned. Only available for level monitors

histogram_autoscale(*ex0=False*)
used by histogram_print to autoscale may be overridden.

Parameters **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes

Returns **bin_width, lowerbound, number_of_bins**

Return type tuple

maximum(*ex0=False*)
maximum of tallied values

Parameters **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes

Returns **maximum**

Return type float

mean(*ex0=False*)
mean of tallied values

Parameters **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes

Returns **mean**

Return type float

Note: If weights are applied , the weighted mean is returned

median(*ex0=False*, *interpolation='linear'*)
median of tallied values

Parameters

- **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes
- **interpolation** (*str*) – Default: 'linear'

  For non weighted monitors: This optional parameter specifies the interpolation method to use when the 50% percentile lies between two data points i < j: 'linear': i + (j - i) * fraction, where fraction is the fractional part of the index surrounded by i and j. (default for monitors that are not weighted not level\n| 'lower': i. 'higher': j. (default for weighted and level monitors) 'nearest': i or j, whichever is nearest. 'midpoint': (i + j) / 2.

  For weighted and level monitors: This optional parameter specifies the interpolation method to use when the 50% percentile corresponds exactly to two data points i and j 'linear': (i + j) /2 'lower': i. 'higher': j 'midpoint': (i + j) / 2.

Returns **median (50% percentile)**

Return type float

merge(**monitors*, ***kwargs*)
merges this monitor with other monitor(s)

Parameters

- **monitors** (*sequence*) – zero of more monitors to be merged to this monitor
- **name** (*str*) – name of the merged monitor default: name of this monitor + ".merged"

Returns **merged monitor**

Return type *Monitor*

Note: Level monitors can only be merged with level monitors Non level monitors can only be merged with non level monitors Only monitors with the same type can be merged If no monitors are specified, a copy is created. For level monitors, merging means summing the available x-values\n|

minimum(*ex0=False*)
minimum of tallied values

Parameters **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes

Returns **minimum**

Return type float

monitor(*value=None*)

enables/disables monitor

Parameters **value** (*bool*) – if True, monitoring will be on. if False, monitoring is disabled if omitted, no change

Returns **True, if monitoring enabled. False, if not**

Return type bool

multiply(*scale=1*, *name=None*)

makes a monitor with all x-values multiplied with scale

Parameters

- **scale** (*float*) – scale to be applied
- **name** (*str*) – name of the multiplied monitor default: name of this monitor

Returns **multiplied monitor**

Return type *Monitor*

Note: Only non level monitors with type float can be multiplied

name(*value=None*)

Parameters **value** (*str*) – new name of the monitor if omitted, no change

Returns **Name of the monitor**

Return type str

Note: base_name and sequence_number are not affected if the name is changed

number_of_entries(*ex0=False*)

count of the number of entries

Parameters **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes

Returns **number of entries**

Return type int

Note: Not available for level monitors

number_of_entries_zero()

count of the number of zero entries

Returns **number of zero entries**

Return type int

Note: Not available for level monitors

percentile(*q*, *ex0=False*, *interpolation='linear'*)

q-th percentile of tallied values

Parameters

- **q** (*float*) – percentage of the distribution values <0 are treated a 0 values >100 are treated as 100
- **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes
- **interpolation** (*str*) – Default: 'linear'

  For non weighted monitors: This optional parameter specifies the interpolation method to use when the desired percentile lies between two data points i < j: 'linear': i + (j - i) * fraction, where fraction is the fractional part of the index surrounded by i and j. (default for monitors that are not weighted not level|n| 'lower': i. 'higher': j. (default for weighted and level monitors) 'nearest': i or j, whichever is nearest. 'midpoint': (i + j) / 2.

  For weighted and level monitors: This optional parameter specifies the interpolation method to use when the percentile corresponds exactly to two data points i and j 'linear': (i + j) /2 'lower': i. 'higher': j 'midpoint': (i + j) / 2.

Returns **q-th percentile** – 0 returns the minimum, 50 the median and 100 the maximum

Return type float

print_histogram(*number_of_bins=None*, *lowerbound=None*, *bin_width=None*, *values=False*, *ex0=False*, *as_str=False*, *file=None*, *sort_on_weight=False*, *sort_on_duration=False*, *sort_on_value=False*)

print monitor statistics and histogram

Parameters

- **number_of_bins** (*int*) – number of bins default: 30 if <0, also the header of the histogram will be surpressed
- **lowerbound** (*float*) – first bin default: 0
- **bin_width** (*float*) – width of the bins default: 1
- **values** (*bool*) – if False (default), bins will be used if True, the individual values will be shown (in alphabetical order). in that case, no cumulative values will be given
- **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes

as_str: bool if False (default), print the histogram if True, return a string containing the histogram

file: file if None(default), all output is directed to stdout otherwise, the output is directed to the file

sort_on_weight [bool] if True, sort the values on weight first (largest first), then on the values itself|n| if False, sort the values on the values itself False is the default for non level monitors. Not permitted for level monitors.

sort_on_duration [bool] if True, sort the values on duration first (largest first), then on the values itself|n| if False, sort the values on the values itself False is the default for level monitors. Not permitted for non level monitors.

sort sort_on_weight [bool] if True, sort the values on weight first (largest first), then on the values itself|n| if False (default), sort the values on the values itself Not permitted for level monitors.

sort_on_duration [bool] if True, sort the values on duration first (largest first), then on the values itself|n| if False (default), sort the values on the values itself Not permitted for non level monitors.

sort_on_value [bool] if True, sort on the values. if False (default), no sorting will take place, unless values is an iterable, in which case sorting will be done on the values anyway.

Returns **histogram (if as_str is True)**

Return type str

Note: If number_of_bins, lowerbound and bin_width are omitted, the histogram will be autoscaled, with a maximum of 30 classes.

print_histograms (*number_of_bins=None*, *lowerbound=None*, *bin_width=None*, *values=False*, *ex0=False*, *as_str=False*, *file=None*)
print monitor statistics and histogram

Parameters

- **number_of_bins** (*int*) – number of bins default: 30 if <0, also the header of the histogram will be surpressed
- **lowerbound** (*float*) – first bin default: 0
- **bin_width** (*float*) – width of the bins default: 1
- **values** (*bool*) – if False (default), bins will be used if True, the individual values will be shown (sorted on the value). in that case, no cumulative values will be given
- **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes
- **as_str** (*bool*) – if False (default), print the histogram if True, return a string containing the histogram
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **histogram (if as_str is True)**

Return type str

Note: If number_of_bins, lowerbound and bin_width are omitted, the histogram will be autoscaled, with a maximum of 30 classes. Exactly same functionality as Monitor.print_histogram()

print_statistics (*show_header=True*, *show_legend=True*, *do_indent=False*, *as_str=False*, *file=None*)
print monitor statistics

Parameters

- **show_header** (*bool*) – primarily for internal use
- **show_legend** (*bool*) – primarily for internal use
- **do_indent** (*bool*) – primarily for internal use
- **as_str** (*bool*) – if False (default), print the statistics if True, return a string containing the statistics
- **file** (*file*) – if None (default), all output is directed to stdout otherwise, the output is directed to the file

Returns **statistics (if as_str is True)**

Return type str

register (*registry*)
registers the monitor in the registry

Parameters **registry** (*list*) – list of (to be) registered objects

Returns **monitor (self)**

Return type *Monitor*

Note: Use Monitor.deregister if monitor does not longer need to be registered.

---

rename (*value=None*)

Parameters **value** (*str*) – new name of the monitor if omitted, no change

Returns **self**

Return type monitor

Note: in contrast to name(), this method returns itself, so can used to chain, e.g. (m0 + m1 + m2+ m3).rename('m0-m3').print_histograms() m0[1000 : 2000].rename('m between t=1000 and t=2000').print_histograms()

---

reset (*monitor=None*, *stats_only=None*)
resets monitor

Parameters

- **monitor** (*bool*) – if True, monitoring will be on. if False, monitoring is disabled if omitted, no change of monitoring state
- **stats_only** (*bool*) – if True, only statistics will be collected (using less memory, but also less functionality) if False, full functionality if omittted, no change of stats_only

reset_monitors (*monitor=None*, *stats_only=None*)
resets monitor

Parameters

- **monitor** (*bool*) – if True (default), monitoring will be on. if False, monitoring is disabled if omitted, the monitor state remains unchanged
- **stats_only** (*bool*) – if True, only statistics will be collected (using less memory, but also less functionality) if False, full functionality if omittted, no change of stats_only

Note: Exactly same functionality as Monitor.reset()

sequence_number()

Returns sequence_number of the monitor – (the sequence number at initialization) normally this will be the integer value of a serialized name, but also non serialized names (without a dot or a comma at the end) will be numbered)

Return type int

setup()

called immediately after initialization of a monitor.

by default this is a dummy method, but it can be overridden.

only keyword arguments are passed

slice(*start=None*, *stop=None*, *modulo=None*, *name=None*)

slices this monitor (creates a subset)

Parameters

- **start** (*float*) – if modulo is not given, the start of the slice if modulo is given, this is indicates the slice period start (modulo modulo)
- **stop** (*float*) – if modulo is not given, the end of the slice if modulo is given, this is indicates the slice period end (modulo modulo) note that stop is excluded from the slice (open at right hand side)
- **modulo** (*float*) – specifies the distance between slice periods if not specified, just one slice subset is used.
- **name** (*str*) – name of the sliced monitor default: name of this monitor + ".sliced"

Returns sliced monitor

Return type *Monitor*

Note: It is also possible to use square bracktets to slice, like m[0:1000].

start_time()

Returns Start time of the monitor – either the time of creation or latest reset

Return type float

`std`(*ex0=False*)

standard deviation of tallied values

Parameters **`ex0`** (*`bool`*) – if False (default), include zeroes. if True, exclude zeroes

Returns **standard deviation**

Return type float

Note: If weights are applied, the weighted standard deviation is returned

`t`()

get the time of last tally of a level monitor

Getter gets the time of the last tallied value : float

Note: t is only available for level monitors t is available even if the monitor is turned off

`tally`(*value*, *weight=1*)

Parameters

- **`x`** (*`any, preferably int, float or translatable into int or float`*) – value to be tallied
- **`weight`** (*`float`*) – weight to be tallied default : 1

`to_days`(*name=None*)

makes a monitor with all x-values converted to days

Parameters **`name`** (*`str`*) – name of the converted monitor default: name of this monitor

Returns **converted monitor**

Return type *Monitor*

Note: Only non level monitors with type float can be converted. It is required that a time_unit is defined for the environment.

`to_hours`(*name=None*)

makes a monitor with all x-values converted to hours

Parameters **name** (*str*) – name of the converted monitor default: name of this monitor

Returns **converted monitor**

Return type *Monitor*

Note: Only non level monitors with type float can be converted. It is required that a time_unit is defined for the environment.

to_microseconds(*name=None*)
makes a monitor with all x-values converted to microseconds

Parameters **name** (*str*) – name of the converted monitor default: name of this monitor

Returns **converted monitor**

Return type *Monitor*

Note: Only non level monitors with type float can be converted. It is required that a time_unit is defined for the environment.

to_milliseconds(*name=None*)
makes a monitor with all x-values converted to milliseconds

Parameters **name** (*str*) – name of the converted monitor default: name of this monitor

Returns **converted monitor**

Return type *Monitor*

Note: Only non level monitors with type float can be converted. It is required that a time_unit is defined for the environment.

to_minutes(*name=None*)
makes a monitor with all x-values converted to minutes

Parameters **name** (*str*) – name of the converted monitor default: name of this monitor

Returns **converted monitor**

Return type *Monitor*

Note: Only non level monitors with type float can be converted. It is required that a time_unit is defined for the environment.

to_seconds (*name=None*)

makes a monitor with all x-values converted to seconds

Parameters **name** (*str*) – name of the converted monitor default: name of this monitor

Returns **converted monitor**

Return type *Monitor*

Note: Only non level monitors with type float can be converted. It is required that a time_unit is defined for the environment.

to_time_unit (*time_unit*, *name=None*)

makes a monitor with all x-values converted to the specified time unit

Parameters

- **time_unit** (*str*) – Supported time_units: “years”, “weeks”, “days”, “hours”, “minutes”, “seconds”, “milliseconds”, “microseconds”
- **name** (*str*) – name of the converted monitor default: name of this monitor

Returns **converted monitor**

Return type *Monitor*

Note: Only non level monitors with type float can be converted. It is required that a time_unit is defined for the environment.

to_weeks (*name=None*)

makes a monitor with all x-values converted to weeks

Parameters **name** (*str*) – name of the converted monitor default: name of this monitor

Returns **converted monitor**

Return type *Monitor*

Note: Only non level monitors with type float can be converted. It is required that a time_unit is defined for the environment.

to_years(*name=None*)
makes a monitor with all x-values converted to years

Parameters **name** (*str*) – name of the converted monitor default: name of this monitor

Returns **converted monitor**

Return type *Monitor*

Note: Only non level monitors with type float can be converted. It is required that a time_unit is defined for the environment.

---

tx(*ex0=False*, *exoff=False*, *force_numeric=False*, *add_now=True*)
tuple of array with timestamps and array/list with x-values

Parameters

- **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes
- **exoff** (*bool*) – if False (default), include self.off. if True, exclude self.off's non level monitors will return all values, regardless of exoff
- **force_numeric** (*bool*) – if True (default), convert non numeric tallied values numeric if possible, otherwise assume 0 if False, do not interpret x-values, return as list if type is list
- **add_now** (*bool*) – if True (default), the last tallied x-value and the current time is added to the result if False, the result ends with the last tallied value and the time that was tallied non level monitors will never add now

Returns **array with timestamps and array/list with x-values**

Return type tuple

Note: The value self.off is stored when monitoring is turned off The timestamps are not corrected for any reset_now() adjustment.

---

value
get/set the value of a level monitor

Getter gets the last tallied value : any (often float)

Setter equivalent to m.tally()

Note: value is only available for level monitors value is available even if the monitor is turned off

---

value_duration(*value*)
total duration of tallied values equal to value or in value

Parameters **value** (*any*) – if list, tuple or set, check whether the tallied value is in value otherwise, check whether the tallied value equals the given value

Returns **total of duration of tallied values in value or equal to value**

Return type float

Note: Not available for non level monitors

value_number_of_entries(*value*)
count of the number of tallied values equal to value or in value

Parameters **value** (*any*) – if list, tuple or set, check whether the tallied value is in value otherwise, check whether the tallied value equals the given value

Returns **number of tallied values in value or equal to value**

Return type int

Note: Not available for level monitors

value_weight(*value*)
total weight of tallied values equal to value or in value

Parameters **value** (*any*) – if list, tuple or set, check whether the tallied value is in value otherwise, check whether the tallied value equals the given value

Returns **total of weights of tallied values in value or equal to value**

Return type int

Note: Not available for level monitors

values(*ex0=False*, *force_numeric=False*, *sort_on_weight=False*, *sort_on_duration=False*)

Parameters

- **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes
- **force_numeric** (*bool*) – if True, convert non numeric tallied values numeric if possible, otherwise assume 0 if False (default), do not interpret x-values, return as list if type is list
- **sort_on_weight** (*bool*) – if True, sort the values on weight first (largest first), then on the values itself|n| if False, sort the values on the values itself False is the default for non level monitors. Not permitted for level monitors.
- **sort_on_duration** (*bool*) – if True, sort the values on duration first (largest first), then on the values itself|n| if False, sort the values on the values itself False is the default for level monitors. Not permitted for non level monitors.

Returns **all tallied values**

Return type array/list

weight(*ex0=False*)
sum of weights

Parameters **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes

Returns **sum of weights**

Return type float

Note: Not available for level monitors

weight_zero()
sum of weights of zero entries

Returns **sum of weights of zero entries**

Return type float

Note: Not available for level monitors

x(*ex0=False*, *force_numeric=True*)
array/list of tallied values

Parameters

- **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes

- **`force_numeric`** (*`bool`*) – if True (default), convert non numeric tallied values numeric if possible, otherwise assume 0 if False, do not interpret x-values, return as list if type is any (list)

Returns **all tallied values**

Return type array/list

Note: Not available for level monitors. Use xduration(), xt() or tx() instead.

`x_map` (*func*, *monitors=[]*, *name=None*)
maps a function to the x-values of the given monitors (static method)

Parameters

- **`func`** (*`function`*) – a function that accepts n x-values, where n is the number of monitors note that the function will not be called during the time any of the monitors is off
- **`monitors`** (*`list/tuple of additional monitors`*) – monitor(s) to be mapped only allowed for level monitors-
- **`name`** (*`str`*) – name of the mapped monitor default: “mapped”

Returns **mapped monitor**

Return type *Monitor*, type ‘any’

`xduration` (*ex0=False*, *force_numeric=True*)
array/list of tallied values

Parameters

- **`ex0`** (*`bool`*) – if False (default), include zeroes. if True, exclude zeroes
- **`force_numeric`** (*`bool`*) – if True (default), convert non numeric tallied values numeric if possible, otherwise assume 0 if False, do not interpret x-values, return as list if type is list

Returns **all tallied values**

Return type array/list

Note: not available for non level monitors

---

xt (*ex0=False*, *exoff=False*, *force_numeric=True*, *add_now=True*)
tuple of array/list with x-values and array with timestamp

Parameters

- **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes
- **exoff** (*bool*) – if False (default), include self.off. if True, exclude self.off's non level monitors will return all values, regardless of exoff
- **force_numeric** (*bool*) – if True (default), convert non numeric tallied values numeric if possible, otherwise assume 0 if False, do not interpret x-values, return as list if type is list
- **add_now** (*bool*) – if True (default), the last tallied x-value and the current time is added to the result if False, the result ends with the last tallied value and the time that was tallied non level monitors will never add now if now is <= last tallied value, nothing will be added, even if add_now is True

Returns array/list with x-values and array with timestamps

Return type tuple

Note: The value self.off is stored when monitoring is turned off The timestamps are not corrected for any reset_now() adjustment.

xweight (*ex0=False*, *force_numeric=True*)
array/list of tallied values

Parameters

- **ex0** (*bool*) – if False (default), include zeroes. if True, exclude zeroes
- **force_numeric** (*bool*) – if True (default), convert non numeric tallied values numeric if possible, otherwise assume 0 if False, do not interpret x-values, return as list if type is list

Returns all tallied values

Return type array/list

Note: not available for level monitors

14.9 Queue

class salabim.**Queue**(*name=None*, *monitor=True*, *fill=None*, *capacity=inf*, *env=None*, **args*, ***kwargs*)

Queue object

Parameters

- **fill** (`Queue, list or tuple`) – fill the queue with the components in fill if omitted, the queue will be empty at initialization
- **name** (*str*) – name of the queue if the name ends with a period (.), auto serializing will be applied if the name end with a comma, auto serializing starting at 1 will be applied if omitted, the name will be derived from the class it is defined in (lowercased)
- **capacity** (*float*) – mximum number of components the queue can contain. if exceeded, a QueueFullError will be raised default: inf
- **monitor** (*bool*) – if True (default) , both length and length_of_stay are monitored if False, monitoring is disabled.
- **env** (`Environment`) – environment where the queue is defined if omitted, default_env will be used

add(*component*)

adds a component to the tail of a queue

Parameters component (`Component`) – component to be added to the tail of the queue may not be member of the queue yet

Note: the priority will be set to the priority of the tail of the queue, if any or 0 if queue is empty This method is equivalent to append()

add_at_head(*component*)

adds a component to the head of a queue

Parameters component (`Component`) – component to be added to the head of the queue may not be member of the queue yet

Note: the priority will be set to the priority of the head of the queue, if any or 0 if queue is empty

add_behind(*component*, *poscomponent*)

adds a component to a queue, just behind a component

Parameters

- **component** (`Component`) – component to be added to the queue may not be member of the queue yet
- **poscomponent** (`Component`) – component behind which component will be inserted must be member of the queue

Note: the priority of component will be set to the priority of poscomponent

add_in_front_of(*component*, *poscomponent*)

adds a component to a queue, just in front of a component

Parameters

- **component** (`Component`) – component to be added to the queue may not be member of the queue yet
- **poscomponent** (`Component`) – component in front of which component will be inserted must be member of the queue

Note: the priority of component will be set to the priority of poscomponent

add_sorted(*component*, *priority*)

adds a component to a queue, according to the priority

Parameters

- **component** (`Component`) – component to be added to the queue may not be member of the queue yet
- **priority** (*type that can be compared with other priorities in the queue*) – priority in the queue

Note: The component is placed just before the first component with a priority > given priority

all_monitors()

returns all monitors belonging to the queue

Returns all monitors

Return type tuple of monitors

animate(**args*, ***kwargs*)

Animates the components in the queue.

Parameters

- **x** (*float*) – x-position of the first component in the queue default: 50
- **y** (*float*) – y-position of the first component in the queue default: 50

- **direction** (*str*) – if "w", waiting line runs westwards (i.e. from right to left) if "n", waiting line runs northeards (i.e. from bottom to top) if "e", waiting line runs eastwards (i.e. from left to right) (default) if "s", waiting line runs southwards (i.e. from top to bottom) if "t", waiting line runs follows given trajectory
- **trajectory** (*Trajectory*) – trajectory to be followed if direction == "t"
- **reverse** (*bool*) – if False (default), display in normal order. If True, reversed.
- **max_length** (*int*) – maximum number of components to be displayed
- **xy_anchor** (*str*) – specifies where x and y are relative to possible values are (default: sw): `nw n ne w c e sw s se`
- **id** (*any*) – the animation works by calling the animation_objects method of each component, optionally with id. By default, this is self, but can be overriden, particularly with the queue
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)

Returns **reference to AnimationQueue object**

Return type AnimationQueue

Note: It is recommended to use sim.AnimateQueue instead

All measures are in screen coordinates

All parameters, apart from queue and arg can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: title - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

animate3d(**args*, ***kwargs*)

Animates the components in the queue in 3D.

Parameters

- **x** (*float*) – x-position of the first component in the queue default: 0
- **y** (*float*) – y-position of the first component in the queue default: 0
- **z** (*float*) – z-position of the first component in the queue default: 0
- **direction** (*str*) – if "x+", waiting line runs in positive x direction (default) if "x-", waiting line runs in negative x direction if "y+", waiting line runs in positive y direction if "y-", waiting line runs in negative y direction if "z+", waiting line runs in positive z direction if "z-", waiting line runs in negative z direction

- **reverse** (*bool*) – if False (default), display in normal order. If True, reversed.
- **max_length** (*int*) – maximum number of components to be displayed
- **layer** (*int*) – layer (default 0)
- **id** (*any*) – the animation works by calling the animation_objects method of each component, optionally with id. By default, this is self, but can be overriden, particularly with the queue
- **arg** (*any*) – this is used when a parameter is a function with two parameters, as the first argument or if a parameter is a method as the instance default: self (instance itself)

Returns **reference to Animation3dQueue object**

Return type Animation3dQueue

Note: It is recommended to use sim.AnimatedQueue instead

All parameters, apart from queue and arg can be specified as: - a scalar, like 10 - a function with zero arguments, like lambda: title - a function with one argument, being the time t, like lambda t: t + 10 - a function with two parameters, being arg (as given) and the time, like lambda comp, t: comp.state - a method instance arg for time t, like self.state, actually leading to arg.state(t) to be called

append(*component*)
appends a component to the tail of a queue

Parameters **component** (Component) – component to be appened to the tail of the queue may not be member of the queue yet

Note: the priority will be set to the priority of the tail of the queue, if any or 0 if queue is empty This method is equivalent to add()

arrival_rate(*reset=False*)
returns the arrival rate When the queue is created, the registration is reset.

Parameters **reset** (*bool*) – if True, number_of_arrivals is set to 0 since last reset and the time of the last reset to now default: False ==> no reset

Returns **arrival rate** – number of arrivals since last reset / duration since last reset nan if duration is zero

Return type float

base_name()

Returns **base name of the queue (the name used at initialization)**

Return type str

clear()
empties a queue

removes all components from a queue

component_with_name(*txt*)
returns a component in the queue according to its name

Parameters **txt** (*str*) – name of component to be retrieved

Returns **the first component in the queue with name txt** – returns None if not found

Return type Component

copy(*name=None*, *copy_capacity=False*, *monitor=<function Queue.monitor>*)
returns a copy of a queue

Parameters

- **name** (*str*) – name of the new queue if omitted, "copy of " + self.name()
- **monitor** (*bool*) – if True, monitor the queue if False (default), do not monitor the queue
- **copy_capacity** (*bool*) – if True, the capacity will be copied if False (default), the resulting queue will always be unrestricted

Returns **queue with all elements of self**

Return type *Queue*

Note: The priority will be copied from original queue. Also, the order will be maintained.

count(*component*)
component count

Parameters **component** (Component) – component to count

Returns

Return type number of occurences of component in the queue

Note: The result can only be 0 or 1

departure_rate(*reset=False*)

returns the departure rate When the queue is created, the registration is reset.

Parameters **reset** (*bool*) – if True, number_of_departures is set to 0 since last reset and the time of the last reset to now default: False ==> no reset

Returns **departure rate** – number of departures since last reset / duration since last reset nan if duration is zero

Return type float

deregister(*registry*)

deregisters the queue in the registry

Parameters **registry** (*list*) – list of registered queues

Returns **queue (self)**

Return type *Queue*

difference(*q*, *name=None*, *monitor=<function Queue.monitor>*)

returns the difference of two queues

Parameters

- **q** (*Queue*) – queue to be 'subtracted' from self
- **name** (*str*) – name of the new queue if omitted, self.name() - q.name()
- **monitor** (*bool*) – if True, monitor the queue if False (default), do not monitor the queue

Returns

Return type queue containing all elements of self that are not in q

Note: the priority will be copied from the original queue. Also, the order will be maintained. Alternatively, the more pythonic - operator is also supported, e.g. q1 - q2

extend(*source*, *clear_source=False*)

extends the queue with components of source that are not already in self (at the end of self)

Parameters

- **source** (*queue, list or tuple*) –
- **clear_source** (*bool*) – if False (default), the elements will remain in source if True, source will be cleared, so effectively moving all elements in source to self. If source is not a queue, but a list or tuple, the clear_source flag may not be set.

Returns

Return type None

Note: The components in source added to the queue will get the priority of the tail of self.

head()

Returns the head component of the queue, if any. None otherwise

Return type *Component*

Note: q[0] is a more Pythonic way to access the head of the queue

index(*component*)

get the index of a component in the queue

Parameters **component** (`Component`) – component to be queried does not need to be in the queue

Returns index of component in the queue – 0 denotes the head, returns -1 if component is not in the queue

Return type int

insert(*index*, *component*)

Insert component before index-th element of the queue

Parameters

- **index** (*int*) – component to be added just before index'th element should be >=0 and <=len(self)
- **component** (`Component`) – component to be added to the queue

Note: the priority of component will be set to the priority of the index'th component, or 0 if the queue is empty

intersection(*q*, *name=None*, *monitor=False*)

returns the intersect of two queues

Parameters

- **q** (`Queue`) – queue to be intersected with self

- **name** (*str*) – name of the new queue if omitted, self.name() + q.name()
- **monitor** (*bool*) – if True, monitor the queue if False (default), do not monitor the queue

Returns queue with all elements that are in self and q

Return type *Queue*

Note: the priority will be set to 0 for all components in the resulting queue the order of the resulting queue is as follows: in the same order as in self. Alternatively, the more pythonic & operator is also supported, e.g. q1 & q2

monitor(*value*)

enables/disables monitoring of length_of_stay and length

Parameters value (*bool*) – if True, monitoring will be on. if False, monitoring is disabled

Note: it is possible to individually control monitoring with length_of_stay.monitor() and length.monitor()

move(*name=None*, *monitor=<function Queue.monitor>*, *copy_capacity=False*)

makes a copy of a queue and empties the original

Parameters

- **name** (*str*) – name of the new queue
- **monitor** (*bool*) – if True, monitor the queue if False (default), do not monitor the yqueue
- **copy_capacity** (*bool*) – if True, the capacity will be copied if False (default), the new queue will always be unrestricted

Returns

- **queue containing all elements of self** (*Queue*)
- *the capacity of the original queue will not be changed*

Note: Priorities will be kept self will be emptied

name(*value=None*)

Parameters value (*str*) – new name of the queue if omitted, no change

Returns **Name of the queue**

Return type str

Note: base_name and sequence_number are not affected if the name is changed All derived named are updated as well.

pop (*index=None*)
removes a component by its position (or head)

Parameters **index** (*int*) – index-th element to remove, if any if omitted, return the head of the queue, if any

Returns **The i-th component or head** – None if not existing

Return type *Component*

predecessor (*component*)
predecessor in queue

Parameters **component** (`Component`) – component whose predecessor to return must be member of the queue

Returns **predecessor of component, if any** – None otherwise.

Return type Component

print_histograms (*exclude=(), as_str=False, file=None*)
prints the histograms of the length and length_of_stay monitor of the queue

Parameters

- **exclude** (*tuple or list*) – specifies which monitors to exclude default: ()
- **as_str** (*bool*) – if False (default), print the histograms if True, return a string containing the histograms
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **histograms (if as_str is True)**

Return type str

print_info (*as_str=False, file=None*)
prints information about the queue

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info

- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

print_statistics (*as_str=False*, *file=None*)
prints a summary of statistics of a queue

Parameters

- **as_str** (*bool*) – if False (default), print the statistics if True, return a string containing the statistics
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **statistics (if as_str is True)**

Return type str

register (*registry*)
registers the queue in the registry

Parameters **registry** (*list*) – list of (to be) registered objects

Returns **queue (self)**

Return type *Queue*

Note: Use Queue.deregister if queue does not longer need to be registered.

remove (*component=None*)
removes component from the queue

Parameters **component** (Component) – component to be removed if omitted, all components will be removed.

Note: component must be member of the queue

rename (*value=None*)

Parameters **value** (*str*) – new name of the queue if omitted, no change

Returns **self**

Return type queue

Note: in contrast to name(), this method returns itself, so can used to chain, e.g. (q0 + q1 + q2 + q3).rename('q0 - q3').print_statistics() (q1 - q0).rename('difference of q1 and q0)').print_histograms()

reset_monitors (*monitor=None*, *stats_only=None*)
resets queue monitor length_of_stay and length

Parameters

- **monitor** (*bool*) – if True, monitoring will be on. if False, monitoring is disabled if omitted, no change of monitoring state
- **stats_only** (*bool*) – if True, only statistics will be collected (using less memory, but also less functionality) if False, full functionality if omittted, no change of stats_only

Note: it is possible to reset individual monitoring with length_of_stay.reset() and length.reset()

sequence_number ()

Returns **sequence_number of the queue** – (the sequence number at initialization) normally this will be the integer value of a serialized name, but also non serialized names (without a dot or a comma at the end) will be numbered)

Return type int

setup ()
called immediately after initialization of a queue.

by default this is a dummy method, but it can be overridden.

only keyword arguments are passed

successor (*component*)
successor in queue

Parameters **component** (Component) – component whose successor to return must be member of the queue

Returns **successor of component, if any** – None otherwise

Return type *Component*

symmetric_difference (*q*, *name=None*, *monitor=<function Queue.monitor>*)
returns the symmetric difference of two queues

Parameters

- **q** (Queue) – queue to be 'subtracted' from self
- **name** (*str*) – name of the new queue if omitted, self.name() - q.name()
- **monitor** (*bool*) – if True, monitor the queue if False (default), do not monitor the queue

Returns

Return type queue containing all elements that are either in self or q, but not in both

---

Note: the priority of all elements will be set to 0 for all components in the new queue. Order: First, elelements in self (in that order), then elements in q (in that order) Alternatively, the more pythonic ^ operator is also supported, e.g. q1 ^ q2

`tail()`

Returns the tail component of the queue, if any. None otherwise

Return type *Component*

Note: q[-1] is a more Pythonic way to access the tail of the queue

---

`union(q, name=None, monitor=False)`

Parameters

- **q** (Queue) – queue to be unioned with self
- **name** (*str*) – name of the new queue if omitted, self.name() + q.name()
- **monitor** (*bool*) – if True, monitor the queue if False (default), do not monitor the queue

Returns queue containing all elements of self and q

Return type *Queue*

Note: the priority will be set to 0 for all components in the resulting queue the order of the resulting queue is as follows: first all components of self, in that order, followed by all components in q that are not in self, in that order. Alternatively, the more pythonic | operator is also supported, e.g. q1 | q2

14.10 Resource

class salabim.**Resource**(*name=None*, *capacity=1*, *initial_claimed_quantity=0*, *anonymous=False*, *preemptive=False*, *honor_only_first=False*, *honor_only_highest_priority=False*, *monitor=True*, *env=None*, **args*, ***kwargs*)

Parameters

- **name** (*str*) – name of the resource if the name ends with a period (.), auto serializing will be applied if the name end with a comma, auto serializing starting at 1 will be applied if omitted, the name will be derived from the class it is defined in (lowercased)
- **capacity** (*float*) – capacity of the resource if omitted, 1
- **initial_claimed_quantity** (*float*) – initial claimed quantity. Only allowed to be non zero for anonymous resources if omitted, 0
- **anonymous** (*bool*) – anonymous specifier if True, claims are not related to any component. This is useful if the resource is actually just a level. if False, claims belong to a component.
- **honor_only_first** (*bool*) – if True, only the first component of requesters will be honoured (default: False)
- **honor_only_highest_priority** (*bool*) – if True, only component with the priority of the first requester will be honoured (default: False) Note: only respected if honor_only_first is False
- **monitor** (*bool*) – if True (default), the requesters queue, the claimers queue, the capacity, the available_quantity and the claimed_quantity are monitored if False, monitoring is disabled.
- **env** (Environment) – environment to be used if omitted, default_env is used

all_monitors()

returns all mononitors belonging to the resource

Returns **all monitors**

Return type tuple of monitors

base_name()

Returns **base name of the resource (the name used at initialization)**

Return type str

claimers()

Returns **queue with all components claiming from the resource** – will be an empty queue for an anonymous resource

Return type *Queue*

deregister(*registry*)
deregisters the resource in the registry

Parameters **registry** (*list*) – list of registered components

Returns **resource (self)**

Return type *Resource*

ispreemptive()

Returns **True if preemptive, False otherwise**

Return type bool

monitor(*value*)
enables/disables the resource monitors

Parameters **value** (*bool*) – if True, monitoring is enabled if False, monitoring is disabled

---

Note: it is possible to individually control monitoring with claimers().monitor() and requesters().monitor(), capacity.monitor(), available_quantity.monitor), claimed_quantity.monitor() or occupancy.monitor()

---

name(*value=None*)

Parameters **value** (*str*) – new name of the resource if omitted, no change

Returns **Name of the resource**

Return type str

Note: base_name and sequence_number are not affected if the name is changed All derived named are updated as well.

---

print_histograms(*exclude=()*, *as_str=False*, *file=None*)
prints histograms of the requesters and claimers queue as well as the capacity, available_quantity and claimed_quantity timstamped monitors of the resource

Parameters

- **exclude** (*tuple or list*) – specifies which queues or monitors to exclude default: ()
- **as_str** (*bool*) – if False (default), print the histograms if True, return a string containing the histograms
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **histograms (if as_str is True)**

Return type str

print_info(*as_str=False*, *file=None*)
prints info about the resource

Parameters

- **as_str** (*bool*) – if False (default), print the info if True, return a string containing the info
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **info (if as_str is True)**

Return type str

print_statistics(*as_str=False*, *file=None*)
prints a summary of statistics of a resource

Parameters

- **as_str** (*bool*) – if False (default), print the statistics if True, return a string containing the statistics
- **file** (*file*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns **statistics (if as_str is True)**

Return type str

register(*registry*)
registers the resource in the registry

Parameters **registry** (*list*) – list of (to be) registered objects

Returns **resource (self)**

Return type *Resource*

Note: Use Resource.deregister if resource does not longer need to be registered.

release(*quantity=None*)
releases all claims or a specified quantity

Parameters **`quantity`** (*`float`*) – quantity to be released if not specified, the resource will be emptied completely for non-anonymous resources, all components claiming from this resource will be released.

Note: quantity may not be specified for a non-anonymous resoure

`requesters`()

Returns **queue containing all components with not yet honored requests**

Return type *Queue*

`reset_monitors`(*monitor=None*, *stats_only=None*)

resets the resource monitors

Parameters

- **`monitor`** (*`bool`*) – if True, monitoring will be on. if False, monitoring is disabled if omitted, no change of monitoring state
- **`stats_only`** (*`bool`*) – if True, only statistics will be collected (using less memory, but also less functionality) if False, full functionality if omittted, no change of stats_only

Note: it is possible to reset individual monitoring with claimers().reset_monitors(), requesters().reset_monitors, capacity.reset(), available_quantity.reset() or claimed_quantity.reset() or occupancy.reset()

`sequence_number`()

Returns **sequence_number of the resource** – (the sequence number at initialization) normally this will be the integer value of a serialized name, but also non serialized names (without a dot or a comma at the end) will be numbered)

Return type int

`set_capacity`(*cap*)

Parameters **`cap`** (*`float or int`*) – capacity of the resource this may lead to honoring one or more requests. if omitted, no change

`setup`()

called immediately after initialization of a resource.

by default this is a dummy method, but it can be overridden.

only keyword arguments are passed

14.11 State

class salabim.**State**(*name=None*, *value=False*, *type='any'*, *monitor=True*, *env=None*, **args*, ***kwargs*)

Parameters

- **name** (*str*) – name of the state if the name ends with a period (.), auto serializing will be applied if the name end with a comma, auto serializing starting at 1 will be applied if omitted, the name will be derived from the class it is defined in (lowercased)
- **value** (*any, preferably printable*) – initial value of the state if omitted, False
- **monitor** (*bool*) – if True (default) , the waiters queue and the value are monitored if False, monitoring is disabled.
- **type** (*str*) – specifies how the state values are monitored. Using a int, uint of float type results in less memory usage and better performance. Note that you should avoid the number not to use as this is used to indicate 'off'
  - "any" (default) stores values in a list. This allows for non numeric values. In calculations the values are forced to a numeric value (0 if not possible) do not use -inf
  - "bool" bool (False, True). Actually integer >= 0 <= 254 1 byte do not use 255
  - "int8" integer >= -127 <= 127 1 byte do not use -128
  - "uint8" integer >= 0 <= 254 1 byte do not use 255
  - "int16" integer >= -32767 <= 32767 2 bytes do not use -32768
  - "uint16" integer >= 0 <= 65534 2 bytes do not use 65535
  - "int32" integer >= -2147483647 <= 2147483647 4 bytes do not use -2147483648
  - "uint32" integer >= 0 <= 4294967294 4 bytes do not use 4294967295
  - "int64" integer >= -9223372036854775807 <= 9223372036854775807 8 bytes do not use -9223372036854775808
  - "uint64" integer >= 0 <= 18446744073709551614 8 bytes do not use 18446744073709551615
  - "float" float 8 bytes do not use -inf
- **env** (Environment) – environment to be used if omitted, default_env is used

all_monitors()

returns all mononitors belonging to the state

Returns all monitors

Return type tuple of monitors

base_name()

Returns **base name of the state (the name used at initialization)**

Return type str

deregister(*registry*)

deregisters the state in the registry

Parameters **registry** (*list*) – list of registered states

Returns **state (self)**

Return type *State*

get()

get value of the state

Returns

value of the state – Instead of this method, the state can also be called directly, like

level = sim.State("level") ... print(level()) print(level.get()) # identical

Return type any

monitor(*value=None*)

enables/disables the state monitors and value monitor

Parameters **value** (*bool*) – if True, monitoring will be on. if False, monitoring is disabled if not specified, no change

---

Note:

it is possible to individually control requesters().monitor(), value.monitor()

---

name(*value=None*)

Parameters **value** (*str*) – new name of the state if omitted, no change

Returns **Name of the state**

Return type str

Note: base_name and sequence_number are not affected if the name is changed All derived named are updated as well.

`print_histograms`(*exclude=(), as_str=False, file=None*)
print histograms of the waiters queue and the value monitor

Parameters

- **`exclude`** (*`tuple or list`*) – specifies which queues or monitors to exclude default: ()
- **`as_str`** (*`bool`*) – if False (default), print the histograms if True, return a string containing the histograms
- **`file`** (*`file`*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns histograms (if as_str is True)

Return type str

`print_info`(*as_str=False, file=None*)
prints info about the state

Parameters

- **`as_str`** (*`bool`*) – if False (default), print the info if True, return a string containing the info
- **`file`** (*`file`*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns info (if as_str is True)

Return type str

`print_statistics`(*as_str=False, file=None*)
prints a summary of statistics of the state

Parameters

- **`as_str`** (*`bool`*) – if False (default), print the statistics if True, return a string containing the statistics
- **`file`** (*`file`*) – if None(default), all output is directed to stdout otherwise, the output is directed to the file

Returns statistics (if as_str is True)

Return type str

`register`(*registry*)
registers the state in the registry

Parameters **registry** (*list*) – list of (to be) registered objetcs

Returns **state (self)**

Return type *State*

Note: Use State.deregister if state does not longer need to be registered.

---

reset(*value=False*)
reset the value of the state

Parameters **value** (*any (preferably printable)*) – if omitted, False if there is a change, the waiters queue will be checked to see whether there are waiting components to be honored

---

Note: This method is identical to set, except the default value is False.

---

reset_monitors(*monitor=None*, *stats_only=None*)
resets the monitor for the state's value and the monitors of the waiters queue

Parameters

- **monitor** (*bool*) – if True, monitoring will be on. if False, monitoring is disabled if omitted, no change of monitoring state
- **stats_only** (*bool*) – if True, only statistics will be collected (using less memory, but also less functionality) if False, full functionality if omittted, no change of stats_only

sequence_number()

Returns **sequence_number of the state** – (the sequence number at initialization) normally this will be the integer value of a serialized name, but also non serialized names (without a dot or a comma at the end) will be numbered)

Return type int

set(*value=True*)
set the value of the state

Parameters **value** (*any (preferably printable)*) – if omitted, True if there is a change, the waiters queue will be checked to see whether there are waiting components to be honored

Note: This method is identical to reset, except the default value is True.

setup()

called immediately after initialization of a state.

by default this is a dummy method, but it can be overridden.

only keyword arguments will be passed

trigger(*value=True*, *value_after=None*, *max=inf*)

triggers the value of the state

Parameters

- **value** (*any (preferably printable)*) – if omitted, True
- **value_after** (*any (preferably printable)*) – after the trigger, this will be the new value. if omitted, return to the the before the trigger.
- **max** (*int*) – maximum number of components to be honored for the trigger value default: inf

Note: The value of the state will be set to value, then at most max waiting components for this state will be honored and next the value will be set to value_after and again checked for possible honors.

waiters()

Returns queue containing all components waiting for this state

Return type *Queue*

14.12 Trajectories

class salabim.**TrajectoryCircle**(*radius*, *x_center=0*, *y_center=0*, *angle0=0*, *angle1=360*, *t0=None*, *vmax=None*, *v0=None*, *v1=None*, *acc=None*, *dec=None*, *orientation=None*, *env=None*)

Circle (arc) trajectory, to be used in Animatexxx through x, y and angle methods

Parameters

- **radius** (*float*) – radius of the circle or arc
- **x_center** (*float*) – x-coordinate of the circle

- **y_center** (*float*) – y-coordinate of the circle
- **angle0** (*float*) – start angle in degrees default: 0
- **angle1** (*float*) – end angle in degrees default: 360
- **t0** (*float*) – time the trajectory should start default: env.now() if not the first in a merged trajectory or AnimateQueue, ignored
- **vmax** (*float*) – maximum speed, i.e. position units per time unit default: 1
- **v0** (*float*) – velocity at start default: vmax
- **v1** (*float*) – velocity at end default: vmax
- **acc** (*float*) – acceleration rate (position units / time units ** 2) default: inf (i.e. no acceleration)
- **dec** (*float*) – deceleration rate (position units / time units ** 2) default: inf (i.e. no deceleration)
- **orientation** (*float*) – default: gives angle in the direction of the movement when calling angle(t) if a one parameter callable, the angle in the direction of the movement will be callled if a float, this orientation will always be returned as angle(t)
- **env** (Environment) – environment where the trajectory is defined if omitted, default_env will be used

duration()

duration of trajectory

Returns duration of trajectory (t1 - t0)

Return type float

in_trajectory(*t*)

is t in trajectory?

Parameters t (*float*) – time at which to evaluate

Returns is t in trajectory?

Return type bool

length

length of traversed trajectory at time t or total length

Parameters t (*float*) – time at which to evaluate lenght. If omitted, total length will be returned

Returns length – length of traversed trajectory at time t or total length if t omitted

Return type float

rendered_polygon(*time_step=1*)
rendered polygon

Parameters **time_step** (*float*) – defines at which point in time the trajectory has to be rendered default : 1

Returns **polygon** – rendered from t0 to t1 with time_step can be used directly in sim.AnimatePoints() or AnimatePolygon()

Return type list of x, y

t0()
start time of trajectory

Returns **start time of trajectory**

Return type float

t1()
end time of trajectory

Returns **end time of trajectory**

Return type float

class salabim.**TrajectoryPolygon**(*polygon*, *t0=None*, *vmax=None*, *v0=None*, *v1=None*, *acc=None*, *dec=None*, *orientation=None*, *spline=None*, *res=50*, *env=None*)
Polygon trajectory, to be used in Animatexxx through x, y and angle methods

Parameters

- **polygon** (*iterable of floats*) – should be like x0, y0, x1, y1, . . .
- **t0** (*float*) – time the trajectory should start default: env.now() if not the first in a merged trajectory or AnimateQueue, ignored
- **vmax** (*float*) – maximum speed, i.e. position units per time unit default: 1
- **v0** (*float*) – velocity at start default: vmax
- **v1** (*float*) – velocity at end default: vmax
- **acc** (*float*) – acceleration rate (position units / time units ** 2) default: inf (i.e. no acceleration)
- **dec** (*float*) – deceleration rate (position units / time units ** 2) default: inf (i.e. no deceleration)
- **orientation** (*float*) – default: gives angle in the direction of the movement when calling angle(t) if a one parameter callable, the angle in the direction of the movement will be callled if a float, this orientation will always be returned as angle(t)
- **spline** (*None or string*) – if None (default), polygon is used as such if 'bezier' (or any string starting with 'b' or 'B', Bézier splining is used if 'catmull_rom' (or any string starting with 'c' or 'C', Catmull-Rom splining is used

- **res** (*int*) – resolution of spline (ignored when no splining is applied)
- **env** (Environment) – environment where the trajectory is defined if omitted, default_env will be used

Notes

bezier and catmull_rom splines require numpy to be installed.

angle(*t*, *_t0=None*)
value of angle (in degrees)

Parameters **t** (*float*) – time at which to evaluate angle

Returns **evaluated angle (in degrees)**

Return type float

duration()
duration of trajectory

Returns **duration of trajectory (t1 - t0)**

Return type float

in_trajectory(*t*)
is t in trajectory?

Parameters **t** (*float*) – time at which to evaluate

Returns **is t in trajectory?**

Return type bool

length(*t=None*, *_t0=None*)
length of traversed trajectory at time t or total length

Parameters **t** (*float*) – time at which to evaluate lenght. If omitted, total length will be returned

Returns **length** – length of traversed trajectory at time t or total length if t omitted

Return type float

rendered_polygon(*time_step=1*)
rendered polygon

Parameters **time_step** (*float*) – defines at which point in time the trajectory has to be rendered default : 1

Returns **polygon** – rendered from t0 to t1 with time_step can be used directly in sim.AnimatePoints() or AnimatePolygon()

Return type list of x, y

t0()
start time of trajectory

Returns **start time of trajectory**

Return type float

t1()
end time of trajectory

Returns **end time of trajectory**

Return type float

x(*t*, *_t0=None*)
value of x

Parameters **t** (*float*) – time at which to evaluate x

Returns **evaluated x**

Return type float

y(*t*, *_t0=None*)
value of y

Parameters **t** (*float*) – time at which to evaluate y

Returns **evaluated y**

Return type float

class salabim.**TrajectoryStandstill**(*xy*, *duration*, *orientation=0*, *t0=None*, *env=None*)
Standstill trajectory, to be used in Animatexxx through x, y and angle methods

Parameters

- **xy** (*tuple or list of 2 floats*) – initial (and final) position. should be like x, y
- **orientation** (*float or callable*) – orientation (angle) in degrees a one parameter callable is also accepted (and will be called with 0) default: 0
- **t0** (*float*) – time the trajectory should start default: env.now() if not the first in a merged trajectory or AnimateQueue, ignored

- **env** (`Environment`) – environment where the trajectory is defined if omitted, default_env will be used

`angle`(*t*, *_t0=None*)
value of angle (in degrees)

Parameters **`t`** (*`float`*) – time at which to evaluate angle

Returns **evaluated angle (in degrees)**

Return type float

`duration`()
duration of trajectory

Returns **duration of trajectory (t1 - t0)**

Return type float

`in_trajectory`(*t*)
is t in trajectory?

Parameters **`t`** (*`float`*) – time at which to evaluate

Returns **is t in trajectory?**

Return type bool

`length`(*t=None*, *_t0=None*)
length of traversed trajectory at time t or total length

Parameters **`t`** (*`float`*) – time at which to evaluate length.

Returns **length** – always 0

Return type float

`rendered_polygon`(*time_step=1*)
rendered polygon

Parameters **`time_step`** (*`float`*) – defines at which point in time the trajectory has to be rendered default : 1

Returns **polygon** – rendered from t0 to t1 with time_step can be used directly in sim.AnimatePoints() or AnimatePolygon()

Return type list of x, y

`t0`()
start time of trajectory

Returns **start time of trajectory**

Return type float

t1()
end time of trajectory

Returns **end time of trajectory**

Return type float

x(*t*, *_t0=None*)
value of x

Parameters **t** (*float*) – time at which to evaluate x

Returns **evaluated x**

Return type float

y(*t*, *_t0=None*)
value of y

Parameters **t** (*float*) – time at which to evaluate y

Returns **evaluated y**

Return type float

14.13 Miscellaneous

salabim.**arange**(*like numpy*)

Parameters

- **start** (*float*) – start value
- **stop** (*: float*) – stop value
- **step** (*float*) – default: 1

Returns

Return type Iterable

Note: If numpy is installed, uses numpy.arange

`salabim.`**`audio_duration`**(*filename*)

duration of a audio file (usually mp3)

Parameters **`filename`** (*`str`*) – must be a valid audio file (usually mp3)

Returns **duration in seconds**

Return type float

Note: Only supported on Windows and Pythonista. On other platform returns 0

`salabim.`**`can_animate`**(*try_only=True*)

Tests whether animation is supported.

Parameters **`try_only`** (*`bool`*) – if True (default), the function does not raise an error when the required modules cannot be imported if False, the function will only return if the required modules could be imported.

Returns **True, if required modules could be imported, False otherwise**

Return type bool

`salabim.`**`can_video`**(*try_only=True*)

Tests whether video is supported.

Parameters **`try_only`** (*`bool`*) – if True (default), the function does not raise an error when the required modules cannot be imported if False, the function will only return if the required modules could be imported.

Returns **True, if required modules could be imported, False otherwise**

Return type bool

`salabim.`**`centered_rectangle`**(*width, height*)

creates a rectangle tuple with a centered rectangle for use with sim.Animate

Parameters

- **`width`** (*`float`*) – width of the rectangle
- **`height`** (*`float`*) – height of the rectangle

`salabim.`**`colornames`**`()`

available colornames

Returns dict with name of color as key, #rrggbb or #rrggbbaa as value

Return type dict

`salabim.`**`random_seed`**`(`*seed=None*, *randomstream=None*, *set_numpy_random_seed=True*`)`

Reseeds a randomstream

Parameters

- **`seed`** (*`hashable object, usually int`*) – the seed for random, equivalent to random.seed() if "*", a purely random value (based on the current time) will be used (not reproducable) if the null string, no action on random is taken if None (the default), 1234567 will be used.
- **`set_numpy_random_seed`** (*`bool`*) – if True (default), numpy.random.seed() will be called with the given seed. This is particularly useful when using External distributions. If numpy is not installed, this parameter is ignored if False, numpy.random.seed is not called.
- **`randomstream`** (*`randomstream`*) – randomstream to be used if omitted, random will be used

`salabim.`**`default_over3d`**`(`*val=None*`)`

Set default_over3d

Parameters **`val`** (*`bool`*) – if not None, set the default_over3d to val

Returns

Return type Current (new) value of default_over3d

`salabim.`**`draw_box3d`**`(`*x_len=1*, *y_len=1*, *z_len=1*, *x=0*, *y=0*, *z=0*, *x_angle=0*, *y_angle=0*, *z_angle=0*, *x_ref=0*, *y_ref=0*, *z_ref=0*, *gl_color=(1, 1, 1)*, *show=True*, *edge_gl_color=(1, 1, 1)*, *show_edge=False*, *shaded=False*, *_show_lids=True*`)`

draws a 3d box (should be added to the event loop by encapsulating with Animate3dBase)

Parameters

- **`x_len`** (*`int, optional`*) – [description], by default 1
- **`y_len`** (*`int, optional`*) – [description], by default 1
- **`z_len`** (*`int, optional`*) – [description], by default 1
- **`x`** (*`int, optional`*) – [description], by default 0
- **`y`** (*`int, optional`*) – [description], by default 0
- **`z`** (*`int, optional`*) – [description], by default 0
- **`x_angle`** (*`int, optional`*) – [description], by default 0

- **y_angle** (*int, optional*) – [description], by default 0
- **z_angle** (*int, optional*) – [description], by default 0
- **x_ref** (*int, optional*) – [description], by default 0
- **y_ref** (*int, optional*) – [description], by default 0
- **z_ref** (*int, optional*) – [description], by default 0
- **gl_color** (*tuple, optional*) – [description], by default (1, 1, 1)
- **show** (*bool, optional*) – [description], by default True
- **edge_gl_color** (*tuple, optional*) – [description], by default (1, 1, 1)
- **show_edge** (*bool, optional*) – [description], by default False
- **shaded** (*bool, optional*) – [description], by default False
- **_show_lids** (*bool, optional*) – [description], by default True

salabim.**draw_cylinder3d**(*x0=0*, *y0=0*, *z0=0*, *x1=1*, *y1=1*, *z1=1*, *gl_color=(1, 1, 1)*, *radius=1*, *number_of_sides=8*, *rotation_angle=0*, *show_lids=True*)

draws a 3d cylinder (should be added to the event loop by encapsulating with Animate3dBase)

Parameters

- **x0** (*int, optional*) – [description], by default 0
- **y0** (*int, optional*) – [description], by default 0
- **z0** (*int, optional*) – [description], by default 0
- **x1** (*int, optional*) – [description], by default 1
- **y1** (*int, optional*) – [description], by default 1
- **z1** (*int, optional*) – [description], by default 1
- **gl_color** (*tuple, optional*) – [description], by default (1, 1, 1)
- **radius** (*int, optional*) – [description], by default 1
- **number_of_sides** (*int, optional*) – [description], by default 8
- **rotation_angle** (*int, optional*) – [description], by default 0
- **show_lids** (*bool, optional*) – [description], by default True

salabim.**draw_line3d**(*x0=0*, *y0=0*, *z0=0*, *x1=1*, *y1=1*, *z1=1*, *gl_color=(1*, *1*, *1)*)

draws a 3d line (should be added to the event loop by encapsulating with Animate3dBase)

Parameters

- **x0** (*int, optional*) – [description], by default 0
- **y0** (*int, optional*) – [description], by default 0
- **z0** (*int, optional*) – [description], by default 0
- **x1** (*int, optional*) – [description], by default 1
- **y1** (*int, optional*) – [description], by default 1
- **z1** (*int, optional*) – [description], by default 1
- **gl_color** (*tuple, optional*) – [description], by default (1, 1, 1)

salabim.**draw_rectangle3d**(*x0=0*, *y0=0*, *z=0*, *x1=1*, *y1=1*, *gl_color=(1*, *1*, *1)*)

draws a 3d rectangle (should be added to the event loop by encapsulating with Animate3dBase)

Parameters

- **x0** (*int, optional*) – [description], by default 0
- **y0** (*int, optional*) – [description], by default 0
- **z** (*int, optional*) – [description], by default 0
- **x1** (*int, optional*) – [description], by default 1
- **y1** (*int, optional*) – [description], by default 1
- **gl_color** (*tuple, optional*) – [description], by default (1, 1, 1)

salabim.**default_env**()

Returns default environment

Return type *Environment*

salabim.**interpolate**(*t*, *t0*, *t1*, *v0*, *v1*)

does linear interpolation

Parameters

- **t** (*float*) – value to be interpolated from

- **t0** (*float*) – f(t0)=v0
- **t1** (*float*) – f(t1)=v1
- **v0** (*float, list or tuple*) – f(t0)=v0
- **v1** (*float, list or tuple*) – f(t1)=v1 if list or tuple, len(v0) should equal len(v1)

Returns **linear interpolation between v0 and v1 based on t between t0 and t1**

Return type float or tuple

Note: Note that no extrapolation is done, so if t<t0 ==> v0 and t>t1 ==> v1 This function is heavily used during animation.

`salabim.`**`interp`**(*x*, *xp*, *fp*, *left=None*, *right=None*)

linear interpolatation

Parameters

- **x** (*float*) – target x-value
- **xp** (*list of float, tuples or lists*) – values on the x-axis
- **fp** (*list of float, tuples of lists*) – values on the y-axis should be same length as p

Returns **interpolated value**

Return type float, tuple or list

Notes

If x < xp[0], fp[0] will be returned If x > xp[-1], fp[-1] will be returned

This function is similar to the numpy interp function.

`salabim.`**`linspace`**(*start*, *stop*, *num*, *endpoint=True*)

like numpy.linspace, but returns a list

Parameters

- **start** (*float*) – start of the space
- **stop** (*float*) – stop of the space
- **num** (*int*) – number of points in the space

- **endpoint** (*bool*) – if True (default), stop is last point in the space if False, space ends before stop

salabim.**over3d**(*val=True*)

context manager to change temporarily default_over3d

Parameters **val** (*bool*) – temporary value of default_over3d default: True

Notes

Use as

```
with over3d():
    an = AnimateText('test')
```

salabim.**random_seed**(*seed=None*, *randomstream=None*, *set_numpy_random_seed=True*)

Reseeds a randomstream

Parameters

- **seed** (*hashable object, usually int*) – the seed for random, equivalent to random.seed() if "*", a purely random value (based on the current time) will be used (not reproducable) if the null string, no action on random is taken if None (the default), 1234567 will be used.
- **set_numpy_random_seed** (*bool*) – if True (default), numpy.random.seed() will be called with the given seed. This is particularly useful when using External distributions. If numpy is not installed, this parameter is ignored if False, numpy.random.seed is not called.
- **randomstream** (*randomstream*) – randomstream to be used if omitted, random will be used

salabim.**regular_polygon**(*radius=1*, *number_of_sides=3*, *initial_angle=0*)

creates a polygon tuple with a regular polygon (within a circle) for use with sim.Animate

Parameters

- **radius** (*float*) – radius of the corner points of the polygon default : 1
- **number_of_sides** (*int*) – number of sides (corners) must be >= 3 default : 3
- **initial_angle** (*float*) – angle of the first corner point, relative to the origin default : 0

salabim.**reset**()

resets global variables

used internally at import of salabim

might be useful for REPLs or for Pythonista

`salabim.`**`searchsorted`**`(`*a*, *v*, *side='left'*`)`
search sorted

Parameters

- **`a`** (*`iterable`*) – iterable to be searched in, must be non descending
- **`v`** (*`float`*) – value to be searched for
- **`side`** (*`string`*) – If 'left' (default) the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of a).

Returns **Index where v should be inserted to maintain order**

Return type int

Note: If numpy is installed, uses numpy.searchstarted

`salabim.`**`show_colornames`**`()`
show (print) all available color names and their value.

`salabim.`**`show_fonts`**`()`
show (print) all available fonts on this machine

`salabim.`**`spec_to_image`**`(`*spec*`)`
convert an image specification to an image

Parameters **`image`** (*`str or PIL.Image.Image`*) – if str: filename of file to be loaded if null string: dummy image will be returned if PIL.Image.Image: return this image untranslated

Returns **image**

Return type PIL.Image.Image

CHAPTER

FIFTEEN

ABOUT

15.1 Who is behind salabim?

Ruud van der Ham is the core developer of salabim. He has a long history in simulation, both in applications and tool building.

It all started in the mid 70's modeling container terminals in Prosim, a package in PL/1 that was inspired by Simula and run on big IBM 360/370 mainframes. In the eighties, Prosim was ported to smaller computers, but at the same time he developed a discrete event simulation tool called Must to run on CP/M machines, later on MSDOS machines, again under PL/1. A bit later, Must was ported to Pascal and was used in many projects. Must was never ported to Windows. Instead, Hans Veeke (Delft University) came with Tomas, a package that is still available and runs under Delphi. End 2016, an easy to use and open source package for a project, preferably in Python, was wanted. Unfortunately, the other Python DES package Simpy (particularly version 3) does not support the essential process interaction methods activate, hold, passivate and standby. First he tried to build a wrapper around Simpy 3, but that didn't work too well.

That was the start of a new package, called salabim. One of the key features of salabim is the powerful animation engine that is heavily inspired by some of his creative projects where every animation object can change position, shape, colour, orientation over time. Although rarely used in normal simulation models, all that functionality is available in salabim. Then, gradually, a lot of functionality was added as well bugs were fixed. Also, the package became available on PyPI and GitHub and the documentation was made available. Large parts of salabim were actually developed on an iPad on the excellent Pythonista platform. Pratically the full functionality is thus available under iOS as well.

15.2 Why is the package called salabim?

The name is derived from the magic words *Sim Salabim*, where Sim is actually short for simulation !

Note that the name should be written in all lowercase, unless it is at the start of a sentence, like a normal noun.

15.3 Contributing and reporting issues

It is very much appreciated to contribute to the salabim, by issuing a pull request or issue on GitHub.

Alternatively, the Google group can be used for this.

15.4 Support

Ruud van der Ham is able and willing to help users with issues with the package and modelling in general.

He is also available for code and model reviews, consultancy, training.

Contact him or other users via the Google group or ruud@salabim.org.

15.5 License

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- search

INDEX