Introduction to Python

Kjartan Halvorsen

February 13, 2017

1 Short introduction to python

1.1 Preliminary

This notebook is part of the repository systemanalys. You can either clone this repository using `git`, or you can download it as a zip-file.

1.2 Installation

1.2.1 Anaconda

In this course we recomment to use the Anaconda python distribution. Make sure to download the Python3 version. The Anaconda distribution contains by default many of the most common python modules for scientific computing. However, it does not come with SimPy. To install this, open the Anaconda prompt (Start -> All Programs -> Anaconda -> Anaconda prompt) and write

```
> pip install simpy
```

You can manage your Anaconda installation and launch programs from the Anaconda Navigator (Start -> All Programs -> Anaconda -> Anaconda Navigator). The anaconda distribution does come with Jupyter, which is a program that lets you work with notebooks (code and documentation interwoven) in your favourite browser. Wonderful to work with!

1.2.2 Jupyter

Writing and running the simulation will be done using Jupyter notebooks. This getting-started-guide is written as a Jupyter notebook. You may be reading this and running the command in jupyter. Or if you are reading the pdf-version you are encouraged to jump over to the Jupyter version right away. You can launch Jupyter notebook from the Anaconda Navigator. This will open a new tab in your default web browser. It shows the current working directory from which you can create folders, files or notebooks. Navigate to the folder containing this notebook (`systemanalys/doc`) and open the notebook Introduction to Python.

1.3 Python variables and datatypes

To do something useful, we need to define *variables* to hold *data*. In python the *type* of the variable is not specified, so a variable name can hold any kind of data. This is called “weak typing”. The most common basic data types are

numbers

```
In [9]: a = 42
```

strings

```
In [10]: s1 = 'system'
         s2 = "system"
```

lists

```
In [11]: b = ['hej', s1, a]
```

In the assignments above, this is what happens in the computer: Memory space is allocated to hold the data (what is on the right hand side of the equal sign), and a variable name is created to reference that data. Also, the variable name is added to the list of current variables. When one variable is assigned to another, the data is not copied (unless it is a simple data type such as a float or integer). Instead we end up with two variables referencing the same piece of data. For instance

```
In [12]: c = b
         c

Out[12]: ['hej', 'system', 42]

In [13]: c[-1] = 24
         print(c)
         print(b)

['hej', 'system', 24]
['hej', 'system', 24]
```

Note the command `c[-1] = 24`. Negative indices in python is a convenient way of accessing elements starting from the end of the list. Note also that the command is (re)assigning the last element of `c` to the value 24, leaving the other elements unchanged. The data referenced by `c` is modified, and the variable `b` which refers to the same data will now show the same change. This is fundamentally different from writing

```
In [17]: c = []
         b

Out[17]: ['hej', 'system', 24]
```

Above, an empty list is created and the variable `c` is then assigned to this empty list instead of whatever it was assigned to earlier. The two variables `c` and `b` no longer reference the same data, and the operation leaves `b` entirely unchanged.

1.4 If, for and while

The `if` construct is used whenever certain lines of code should be executed only when some condition holds. For instance

```
In [18]: if (c == []):
             print("c is empty")

c is empty
```

The block of code (lines of code) to be executed if the `if` statement is true is identified by being further *indented* than the line starting with `if`. There are no curly brackets or keywords marking the start and end of the block. The first line that are equally or less indented than the `if` line ends the block of code to be executed. In the example above, this line is empty.

To repeat some piece of code a known number of times, use a `for` loop:

```
In [20]: for i in range(len(b)):
             print(b[i])

hej
system
24
```

The function `len()` returns the length of the list, and the function `range(n)` returns a list of integers starting at 0 and ending at $n-1$. It may seem a bit unnecessary to create a list of integers to use as indices into `b`. Sometimes the index is actually needed, but more often we loop over the elements more conveniently

```
In [21]: for e in b:
             print(e)

hej
system
24
```

A `while` loop is useful to repeat a number of lines for as long as some condition is true

```
In [22]: x = 7
         while x < 1000:
             x += 7
         x

Out[22]: 1001
```

`while` can also be used to create an eternal loop

```
In [ ]: while True:
            x *= x
        x
```

To interrupt the neverending execution of the above code, look to the Jupyter menu near the top of the browser page and choose Kernel -> Interrupt

1.5 Modules

So far we have entered the code directly into the command window. More commonly, one writes code in text files which are then read and executed by the interpreter. A python file (with ending `.py`) containing code is referred to as a *module*. At start-up, some of these modules are loaded automatically and are available for use in our own code. Other modules must be explicitly imported before the code can be accessed. There is, for instance, a `math` module containing a number of mathematical functions, and 'numpy' for numerical computations and linear algebra. To access functions from these modules we must do

```
In [26]: import math
         import numpy as np
```

then we can use functions such as

```
In [27]: print (math.sin(math.pi))
         print (np.sin(np.pi/2))

1.2246467991473532e-16
1.0
```

other modules of interest include SciPy for scientific computing, matplotlib for plotting and pandas for data analysis. For simulations of discrete event simulations we will use the SimPy module.

1.6 Functions

Functions are fundamental constructs in programming languages. They make code much easier to understand and maintain by separating and encapsulating operations. Consider this simple function that swops the first and last element of a list:

```
In [32]: def swop(alist):
              slask = alist[0]
              alist[0] = alist[-1]
              alist[-1] = slask
               # Think about cases where the function will fail and try to figure out how to make it more

         print(b)
         swop(b)
         print(b)
```

```
[24, 'system', 'hej']
['hej', 'system', 24]
```

The function above did some operations on the input argument, but did not return any value. This is of course possible also:

```
In [36]: def middle(alist):
             ''' Will return the element in the middle of the list.
                 If the length is even, it will return the last element before the middle. '''
             ind = int(len(alist)/2 -1) # since indexing starts at 0
             return alist[ind]

         middle(b)
```

```
Out[36]: 'hej'
```

1.7 Creating your own datatypes

Basically all modern programming languages allows you to create your own datatypes based on the fundamental types that the language provides. Assume that we would like to create a datatype `Customer`. We want to use it as this:

```
In [ ]: c1 = Customer("Pontus", prio=1)
        c2 = Curstomer("Elisa", prio=3)
        queue = [c1, c2]
```

Which gives us a short queue containing two customers. The answer is to define a *class* `Customer` from which we can instantiate (generate) any number of *objects*. The values `name` and `prio` are *attributes* of the object:

```
In [45]: class Customer:
             def __init__(self, name, prio=1):
                 self.name = name
                 self.prio = prio

         c1 = Customer("Pontus", prio=1)
         c2 = Customer("Elisa", prio=3)
         queue = [c1, c2]
         queue
```

```
Out[45]: [<__main__.Customer at 0x7fb9345a1940>, <__main__.Customer at 0x7fb9345a15f8>]
```

The function `__init()__` is called the *constructor*. It is called when a new object is created. The first argument, `self`, is a reference to the actual object under construction. We see from the code that the constructor takes two arguments: the name and the priority of the customer. The priority has a default value of 1, which is used if the customer object is created with one argument only. We may check the name and priority of the customers as

```
In [46]: for c in queue:
             print( "Customer %s has priority %d" % (c.name, c.prio) )
```

```
Customer Pontus has priority 1
Customer Elisa has priority 3
```

The somewhat complicated syntax on the `print` line is actually a quite common way (many programming languages has it) of formatting output that contains numbers. Within the string, the percentage-expressions are placeholders for numbers or strings. The occurance of the placeholders are matched with the elements in the *tuple* following the string (the tuple is the list within parantheses separated from the string by the percentage symbol). A tuple is a fundamental datatype in python. It is a list whose elements cannot be changed after it is constructed.

One important advantage of objects and object-oriented programming is that functions written to work on the objects can be associated with the object itself. These functions are called *methods* or *member functions*. If we have an object, say a `list` object, we can print a list of the methods and attributes for that object with the help of the `dir` function:

```
In [47]: names = ['Evelyn', 'Eirik']
         dir(names)
```

```
Out[47]: ['__add__',
          '__class__',
          '__contains__',
          '__delattr__',
          '__delitem__',
          '__dir__',
          '__doc__',
          '__eq__',
          '__format__',
          '__ge__',
          '__getattribute__',
          '__getitem__',
          '__gt__',
          '__hash__',
          '__iadd__',
          '__imul__',
          '__init__',
          '__init_subclass__',
          '__iter__',
          '__le__',
          '__len__',
          '__lt__',
          '__mul__',
          '__ne__',
          '__new__',
          '__reduce__',
          '__reduce_ex__',
          '__repr__',
          '__reversed__',
          '__rmul__',
          '__setattr__',
          '__setitem__',
          '__sizeof__',
          '__str__',
          '__subclasshook__',
          'append',
          'clear',
          'copy',
          'count',
          'extend',
          'index',
          'insert',
          'pop',
          'remove',
          'reverse',
          'sort']
```

You can use the `help()` function to get help on specific methods. For instance

```
In [48]: help(names.sort)
```

```
Help on built-in function sort:

sort(...) method of builtins.list instance
    L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*
```

This help string tells us that the `sort()` method will sort the list in place, i.e. it will be modified by this method. It will try to sort the list depending on the types of the elements. So sorting a list of strings is

```
In [51]: names = ["Evelyn", "Elisa", "Eirik", "Emilio"]
         print(names)
         names.sort()
         names
```

```
['Evelyn', 'Elisa', 'Eirik', 'Emilio']
```

```
Out[51]: ['Eirik', 'Elisa', 'Emilio', 'Evelyn']
```

1.8 Generators

Generators is a powerful concept in python. It is used to implement the processes in the pseudo-parallell / process-based simulations in SimPy. You should think of generators as functions which generateas and returns elements in a sequence. Let's look at a simple example which generates a sequence of odd numbers:

```
In [54]: def oddnumbers():
           x = 1
           while True:
             yield x
             x = x+2

         og = oddnumbers() # This creates the generator

         print( next(og) ) # Prints 1
         print( next(og) ) # Prints 3

         for i in range(4): # The odd numbers 5, 7, 9, 11
           print( next(og) )
```

```
1
3
5
7
9
11
```

Python recognizes a definition of a generator by the existence of the keyword `yield`. In the example, a generator object is created `og = oddnumbers()`, and then the function `next(og)` is called on the generator object. The first call to `next()` causes the code in the definition of the generator to be executed from the top to the first occurance of `yield`. The statment `yield` is similar to a return statement in a regular function. However, in the next call to `next()`, the execution starts at the first line after the `yield` statement that caused the previous return to the caller. The execution goes on to the next occurance of `yield`. The local variables in the definition of the generator retains there values between calls to `next()`.

If a call to `next()` should not reach a `yield` statement, but instead reach the end of the definition of the generator, then an exception is raised which tells the caller to stop iterating over the generator: There are no more elements in the sequence. The odd numbers are of course an infinite sequence, so it should be possible (in theory) to generate increasingly large odd numbers by calling `next(og)` forever.

1.8.1 A classical example: the Fibonacci numbers

The Fibonacci number series is a sequence of integers defined by the difference equation

$$F(n) = F(n-1) + F(n-2)$$

together with the starting conditions $F(0) = 0$, $F(1) = 1$. Leonardo Fibonacci used the difference equation as a simple model of the dynamics in a population of rabbits. In the model there are no deaths, so at each time step, the number of rabbits is equal to the number of rabbits in the previous time step plus the number of rabbits born since the last time step. Each rabbit has exactly one offspring in each time step, except for an initial newborn-period of one time step. Hence the number of rabbits in the last time step that are mature enough to reproduce equals the size of the population two time steps back in time.

A standard schoolbook implementation of the fibonacci numbers would use recursion to compute the numbers:

```
In [58]: def fib_rec(n):
             if n==1 or n==0:
                 return n
             return fib_rec(n-1)+fib_rec(n-2)

         fib_rec(7)

Out[58]: 13
```

The implementation looks kind of nice, but this type of recursive implementation is not a good idea in a modern high-level language such as python. The reason being that function calls are quite expensive as compared to C. Each call to `fib_rec()` causes two new calls to the function. Also, values are computed twice.

The sequence is much more efficiently implemented with a generator

```
In [60]: def fib():
             """ Generates the sequence if fibonacci numbers """
             fmin1 = 0
             fmin2 = 1
             while True:
                 f = fmin1+fmin2
                 yield f
```

```
            # Update
            (fmin2, fmin1) = (fmin1, f)

    fg = fib()
    [next(fg) for i in range(7)]
```

```
Out[60]: [1, 1, 2, 3, 5, 8, 13]
```

The line `[next(fg) for i in range(7)]` is an example of *list comprehension*. A list is generated from the return values of the function call `next(fg)`. This is done 7 times.

Timing the two implementations of the fibonacci numbers show clearly the difference in efficiency.

```
In [66]: import time
         t = time.time()
         fg = fib()
         fibonacciSequence = [next(fg) for i in range(30)]
         print( time.time() - t)
         t = time.time()
         fib_rec(30)
         print( time.time() - t)
```

```
0.00012254714965820312
0.4177854061126709
```

1.9 Random number generation

Discrete event systems are almost always stochastic systems. To build a model and simulate it, we need sequences of random numbers. A truly random sequence of numbers is actually very difficult to generate. Instead, the computer will generate a completely deterministic sequence of numbers, but whose properties are close to that of a sequence of random numbers. The numbers are therefore called pseudo-random.

Python, as well as most other programming languages used in scientific computing, has a built-in pseudo-number generator. We don't have to implement this ourselves. The pseudo-number generator is part of the random module. It is used as follows

```
In [68]: import random
         random.seed(1315)
         print( [random.gauss(mu=1.0, sigma=0.5) for i in range(3)] )
         random.seed(1314)
         print( [random.gauss(mu=1.0, sigma=0.5) for i in range(3)] )
         random.seed(1315)
         print( [random.gauss(mu=1.0, sigma=0.5) for i in range(3)] )
```

```
[0.6201987142056988, 1.3120253897349055, 0.7844364422139856]
[0.5896986001391438, 1.325007288757538, 1.2354502475094375]
[0.6201987142056988, 1.3120253897349055, 0.7844364422139856]
```

Note that the sequence of the first three numbers are the same after the seed is set to the same number. If you choose not to set a seed, the default behaviour of python is to use the system time to set a seed that will be different every time you run your code. The purpose of setting the seed explicitly is to be able to reproduce simulation experiments.

Use `dir(random)` to see the different methods available to generate random numbers from different distributions. Use `help()` to get helt on any specific method. For instance

```
In [70]: help(random.expovariate)
```

```
Help on method expovariate in module random:

expovariate(lambd) method of random.Random instance
    Exponential distribution.

    lambd is 1.0 divided by the desired mean.  It should be
    nonzero.  (The parameter would be called "lambda", but that is
    a reserved word in Python.)  Returned values range from 0 to
    positive infinity if lambd is positive, and from negative
    infinity to 0 if lambd is negative.
```

1.10 The SimPy module

There are good introductory tutorials, examples and documentation for SimPy. In this section we just want to point out a key concept to understand when implementing simulation models in SimPy.

The *processes* of a discrete event system are implemented as *generators* in python. The defining property of a generator is that it can return a value after a call (by use of the `yield` keyword), and continue execution where it left off at the next call. This is exactly what we need to implement processes that performs some operations and then wait for some event to happen before continuing.

In SimPy, the `yield` statement in the process generators **must** return an event object. The process will continue when the event happens. Let's look at some simple examples.

```
In [107]: import random
          import simpy
```

1.10.1 Example 1 - Reneging customers

Consider the process of a customer that enters a quee, waits for 1.3 time units and then exits the queue. The process step of waiting is implemented by returning a `timeout` event object. The event will occur when the specified time has passed. We will assme that if the customer is still in queue when the time has passed, then she will get impatient and exit the system. Otherwise, we assume that the customer has already been served or is currently getting served.

```
In [108]: def reneging_customer_proc(env, name, patience, queue):
              print( "%s with patience %f enters the queue at time %f" % (name, patience, env.now) )
              queue.append(name) # Customers are identified by name, so all names should be unique
              yield env.timeout(1.3)
              if name in queue:
                  queue.remove(name)
                  print( "%s got impatient and exited the queue at time %f" % (name, env.now) )

          env = simpy.Environment()
          queue = []

          env.process(reneging_customer_proc(env, "Oscar", 1.3, queue))
          env.run()
```

```
Oscar with patience 1.300000 enters the queue at time 0.000000
Oscar got impatient and exited the queue at time 1.300000
```

Customer generator We will need a process that generates customers that enters the system. Typically, the time between arrivals is random. Here we assume the time between arrivals to be exponentially distributed with mean time 1.0.

```
In [109]: def customer_generator_proc(env, numberOfCustomers, timeBetween, queue, newArrivalEvents):
              """ Will generate a fixed number of customers, with random time between arrivals."""
              k = 0
              while k<numberOfCustomers:
                  yield env.timeout( random.expovariate(1.0/timeBetween) )
                  k += 1
                  env.process( reneging_customer_proc(env, name = "Customer-%d" %k, patience = 1.3, queu
                  while newArrivalEvents != []:
                      ev = newArrivalEvents.pop(0)
                      # The newArrivalEvents list contains events that servers are waiting for in order
                      # What they are waiting for is for a new customer to arrive, so trigger the event
                      ev.succeed()
                      print( "Triggering arrival event")

          env = simpy.Environment()
          queue = []
          waitingForArrivalEvents = []
          env.process( customer_generator_proc(env, 8, 0.8, queue, waitingForArrivalEvents) )
          env.run()
```

```
Customer-1 with patience 1.300000 enters the queue at time 1.468228
Customer-2 with patience 1.300000 enters the queue at time 2.243378
Customer-1 got impatient and exited the queue at time 2.768228
Customer-3 with patience 1.300000 enters the queue at time 3.056802
Customer-2 got impatient and exited the queue at time 3.543378
Customer-3 got impatient and exited the queue at time 4.356802
Customer-4 with patience 1.300000 enters the queue at time 5.027282
Customer-4 got impatient and exited the queue at time 6.327282
Customer-5 with patience 1.300000 enters the queue at time 7.311952
Customer-6 with patience 1.300000 enters the queue at time 7.859252
Customer-7 with patience 1.300000 enters the queue at time 8.178111
Customer-8 with patience 1.300000 enters the queue at time 8.350823
Customer-5 got impatient and exited the queue at time 8.611952
Customer-6 got impatient and exited the queue at time 9.159252
Customer-7 got impatient and exited the queue at time 9.478111
Customer-8 got impatient and exited the queue at time 9.650823
```

Service process There is a service process that takes care of the customers in the queue. The time to serve a single customer is exponentially distributed with mean 1.3 time units. The service process takes out customers from the queueu on a first come first served basis (FIFO queue). If there are no customers, the server process will wait for the event that a new customer is arriving.

```
In [110]: def service_proc(env, serviceTime, queue, arrivalEvents):
              while True:
                  while queue != []:
                      customer = queue.pop(0) # Take out the first customer
                      yield env.timeout( random.expovariate(1.0/serviceTime) )
                      print( "%s served at time %f" %(customer, env.now) )
                  newArrivalEv = env.event()
                  arrivalEvents.append(newArrivalEv)
                  print( "Service process waiting for a new arrival" )
                  yield newArrivalEv # Wait for the arrival event to be triggered. This is done in cust

          env = simpy.Environment()
          queue = []
```

```
        waitingForArrivalEvents = []
        env.process( service_proc(env, 1.3, queue, waitingForArrivalEvents) )
        env.process( customer_generator_proc(env, 4, 0.8, queue, waitingForArrivalEvents) )

        env.run()
```

```
Service process waiting for a new arrival
Triggering arrival event
Customer-1 with patience 1.300000 enters the queue at time 2.193581
Customer-1 served at time 2.324042
Service process waiting for a new arrival
Triggering arrival event
Customer-2 with patience 1.300000 enters the queue at time 3.629634
Customer-3 with patience 1.300000 enters the queue at time 3.957301
Customer-2 served at time 4.996579
Customer-4 with patience 1.300000 enters the queue at time 5.474184
Customer-4 got impatient and exited the queue at time 6.774184
Customer-3 served at time 7.580712
Service process waiting for a new arrival
```

1.10.2 Exercise: What is the probability that a customer will leave the system without getting served?

The code for the simulation model above with reneging customers is repeated here below, but without the print statements. Think about how you can record the number of customers that leave without being served in the simulation. Use this to answer the question.

```
In [104]: def reneging_customer_proc(env, name, patience, queue):
              queue.append(name) # Customers are identified by name, so all names should be unique
              yield env.timeout(1.3)
              if name in queue:
                  queue.remove(name)

          def customer_generator_proc(env, numberOfCustomers, timeBetween, queue, newArrivalEvents):
              """ Will generate a fixed number of customers, with random time between arrivals."""
              k = 0
              while k<numberOfCustomers:
                  yield env.timeout( random.expovariate(1.0/timeBetween) )
                  k += 1
                  env.process( reneging_customer_proc(env, name = "Customer-%d" %k, patience = 1.3, que
                  while newArrivalEvents != []:
                      ev = newArrivalEvents.pop(0)
                      # The newArrivalEvents list contains events that servers are waiting for in order
                      # What they are waiting for is for a new customer to arrive, so trigger the event
                      ev.succeed()

          def service_proc(env, serviceTime, queue, arrivalEvents):
              while True:
                  while queue != []:
                      customer = queue.pop(0) # Take out the first customer
                      yield env.timeout( random.expovariate(1.0/serviceTime) )
                  newArrivalEv = env.event()
                  arrivalEvents.append(newArrivalEv)
                  yield newArrivalEv # Wait for the arrival event to be triggered. This is done in cust
```

```
env = simpy.Environment()
queue = []
waitingForArrivalEvents = []
env.process( service_proc(env, 1.3, queue, waitingForArrivalEvents) )
env.process( customer_generator_proc(env, 4, 0.8, queue, waitingForArrivalEvents) )

env.run()
```