PyCh tutorial 4DC10

Version: November 8, 2021

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mechanical Engineering
Eindhoven University of Technology
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands

1 Introduction

This interactive book is about the modeling of the operation of systems, e.g. semiconductor factories, assembly and packaging lines, car manufacturing plants, steel foundries, metal processing shops, beer breweries, health care systems, warehouses, order-picking systems. For a proper functioning of these systems, these systems are controlled by operators and electronic devices, e.g. computers.

During the design process, engineers make use of (analytical) mathematical models, e.g. algebra and probability theory, to get answers about the operation of the system. For complex systems, (numerical) mathematical models are used, and computers perform simulation experiments, to analyze the operation of the system. Simulation studies give answers to questions like:

- What is the throughput of the system?
- What is the effect of set-up time in a machine?
- How will the batch size of an order influence the flow time of the product-items?
- What is the effect of more surgeons in a hospital?

The operation of a system can be described, e.g. in terms of concurrent or parallel operating processes. An example of a system with parallel operating processes is a manufacturing line, with a number of manufacturing machines, where product-items go from machine to machine. A surgery room in a hospital is a system where patients are treated by teams using medical equipment and sterile materials. A biological system can be described by a number of parallel processes, where, e.g. processes transform sugars into water and carbon-dioxide producing energy. In all these examples, processes operate in parallel to complete a task, and to achieve a goal. Concurrency is the dominant aspect in these type of systems, and as a consequence this holds too for their models.

The operating behavior of parallel processes can be described by different formalisms, e.g. automata, Petri-nets or parallel processes. This book uses PyCh, a simulation library for Python specially developed for the course *analysis of production systems*. Modelling systems in PyCh is done in the following way: a system is abstracted into a model, with cooperating processes, where processes are connected to each other via channels. The channels are used for exchanging material and information. Models of the above mentioned examples consist of a number of concurrent processes connected by channels, denoting the flow of products, patients or personnel. In PyCh, communication takes place in a synchronous manner. This means that communication between a sending process, and a receiving process takes place only when both processes are able to communicate. Processes and channels can dynamically be altered.

PyCh has a notation of *simulation time*, which is different from the *real time* that we experience. Simulation time represents the time inside our simulation; we can potentially simulate a process which takes days, months, or years, in just seconds of real time. Processes experience simulation time passing by in two ways: they can delay themselves for a fixed or stochastic period of time, which represents the time it takes to perform their actions. Or they can wait indefinitely untill another process is ready to communicate with them over a channel.

1.1 PyCh in a nutshell

During the past decades, simulation tools have been used with success, for the analysis of a variety of (industrial) systems. PyCh is a simulation library for Python, built on top of SimPy (https://simpy.readthedocs.io/en/latest/index.html), a discrete-event simulation library. PyCh is largely inspired by the simulation language Chi 3 (https://cstweb.wtb.tue.nl/chi/trunk-r9682/) which was also developed at the Eindhoven University of Technology (https://www.tue.nl/). PyCh's features are:

- A system (and its control) is modeled as a collection of parallel running processes, communicating with each other using channels.
- All processes live in a simulation 'environment'. We can execute this environment to simulate its processes.
- Processes do not share data with other processes and channels are synchronous (sending and receiving is always done together at the same time), making reasoning about process behaviour easier.
- Processes and channels are dynamic, new processes can be created as needed, and communication channels can be created or rerouted.
- Time and (quasi-) random number generation distributions are available for modeling behavior of the system in time.
- Finally, PyCh is a Python library, so it can be extended with the many available Python libraries that already exist, such as:
  - Numpy (https://numpy.org/), a scientific computing package
  - Scipy (https://scipy.org/), a scientific computing package for algorithms
  - Matplotlib (https://matplotlib.org/), a library for creating graphs and other visualizations
  - Or any other of the thousands of existing Python libraries

Some of the concepts which use when modelling and simulating in PyCh are explained in the following chapters:

- **Chapter 2: Datatypes**; such as integers, real numbers, strings, lists, sets and dictionaries.
- **Chapter 3: Statements**; such as the assign statement, and the if, while, and for statements.
- **Chapter 4: Functions and classes**; how you create your own functions and (data)classes.
- **Chapter 5: Input and output**; such as printing output and plotting graphs to the screen, reading and writing files.
- **Chapter 6: Modelling stochastic behaviour**; using lambda functions and constant/discrete/continuous distributions.
- **Chapter 7: Processes**; how you define processes, how their behaviour is modelled, and how they interact with each other and the environment.
- **Chapter 8: Channels**; how you define channels, and how processes can use them to communicate.
- **Chapter 9: Buffers**; what are the different types of buffers, and how can you use buffers to store entities that are not ready yet to be processed.
- **Chapter 10: Production lines**; how do you model a production line with different types of servers.

1.2 Python

As mentioned before, the PyCh is a simulation package for the general-purpose programming language Python (https://www.python.org/). Python is one of the most useful languages an engineer can learn; it is *the* most popular of all programming languages, and it it sees widespread use in both the scientific community, and in industry. Python is beginner friendly, it is versatile, it has a simple syntax that mimics

the english language, and one of its main advantages, is that it has thousands of ready to use packages you can import into your own code, such as for example the aforementioned scientific libraries: Numpy (https://numpy.org/), Scipy (https://scipy.org/), and Matplotlib (https://matplotlib.org/).

Most of you will be familiar with Matlab. Python and Matlab share a lot of the same syntax. You can click here (https://realpython.com/matlab-vs-python/) if you are interested in the similarities and differences between these two in for example functionality (https://realpython.com/matlab-vs-python/#matlab-vs-python-comparing-features-and-philosophy) or syntax (https://realpython.com/matlab-vs-python/#syntax-differences-between-matlab-and-python). One of the most important differences is that Python relies on indentation to define the scope of function definitions, and if/while/for statements; no end statement is used. Another big difference is that the numbering in Python is zero-based instead of first-based: the first index in a sequence has index `0`.

For example, the following code in Matlab:

```
def function():
    x = [1, 2]
    if x[1] == 1:
        y = x     % A comment
    end
end
```

Becomes this in Python:

```
def function():
    x = [1, 2]
    if x[0]:
        y = x     # A comment
```

If you have zero knowledge of Python, or any similar progamming languages, we recommend that you take a look at some of the first few chapters of a Python introduction course such as this one (https://www.w3schools.com/python/default.asp).

1.3 Jupyter Notebook

This book is written in Jupyter Notebook (https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html), which you must already know if you are reading this. Jupyter notebook is an open source web-application that allows us to share documents that integrate live code as well as explanatory texts, equations and figures. In this course, we use Jupyter notebook to model and simulate manufacturing systems. As you will see in the tutorial, first a piece of explanatory text about the code can be written and this can be followed by the piece of code. This allows for an interactive form of computing, since you can read about the code, run the code, adjust the code and repeat. This interactive form of computing makes it easier to understand the code. To get more in formation on the possibilities in jupyter notebook and how to use it some links are provided below.

The userface components of jupyter notebook such as the tool bar you see above, are explained here (https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/Notebook%20Basics.html), and for more in-depth information you can check this link (https://jupyter.brynmawr.edu/services/public/dblank/Jupyter%20Notebook%20Users%20Manual.ipynb).

Additionally, here (https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/examples_index.html) you can find examples on all the important topics in jupyter notebook. If you encounter any problems, try troubleshooting with the help of this link (https://jupyter-notebook.readthedocs.io/en/stable/troubleshooting.html).

Every notebook in jupyter notebook is composed of *cells*. A cell can contain text (which is called markdown), or executable code. The most important buttons in Jupyter are shown in Figure 1.1.

Figure 1.1: The Run, interrupt, restart, and re-run buttons.

These buttons are:

1. Saving the notebook
2. Adding a cell
3. Running a cell
4. Interrupting the kernel (all code stops executing)
5. Restarting the kernel (all code that was ran is "forgotten")
6. Restarting the kernel and re-running all cells
7. The dialog button which allows you to set a cell to either Code or Markdown.

1.4 Modelling and simulating with PyCh

In this section we will walk through an example of how we will be modelling and simulating with PyCh in Jupyter Notebook. You do not need to fully understand the code or all the modelling steps we show, they are just there to show what we can do with the PyCh simulation tool. During this course, you will slowly learn how to make these models yourself.

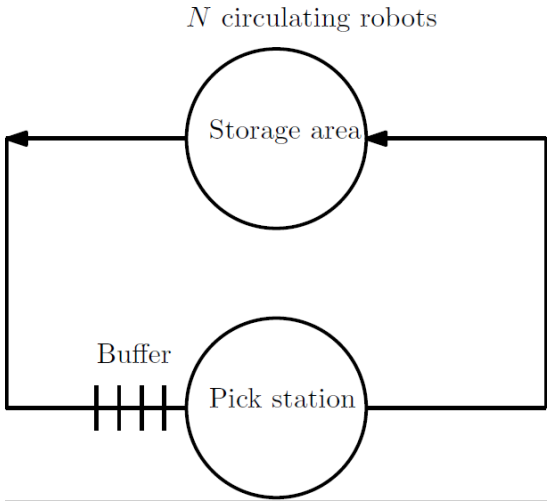
The system we will model is the KIVA system: a new technology in warehousing which uses self-driving robots (so called 'pods') which can pick up and move racks to help in order-picking (shown in Figure 1.1). You might recognize this system from the lecture notes (you can also find a video of the KIVA system here (https://www.youtube.com/watch?v=6KRjuuEVEZs)). In this warehousing sytems, racks, containing items ordered by a customer, are automatically retrieved from the storage area and transported to an order-picking station by these robots. These robots are small autonomous drive units that can carry a rack (by moving under and then lifting the rack). The benefits of such automated warehousing systems are, for example, high throughput capability, flexibility and scalability. By adding more robots, the throughput capacity of handling additional customer orders can be increased in a relatively short time span. The question is: How many robots are needed to achieve a certain target throughput capacity?

A model of the KIVA system is shown in Figure 1.3. The robots pick a rack up at the *storage area*, and bring it to the *pick station*. When they arrive at the pick station, they first arriva at a *buffer*, a queue of robots waiting to be picked. Once the items are picked, the robot returns to the storage aread and the cycle repeats.

Figure 1.2: The KIVA warehouse system. **Figure 1.3: A model of the KIVA system.**

Figure 1.2: The KIVA warehouse system.

Figure 1.3: A model of the KIVA system.



When modelling in Jupyter Notebook, we can create our model step-by-step. There is no need to create the entire model in one go. **Keep in mind, everything that you define (variables, functions, processes, etc.) is not forgotten until you either redefine it, or until you restart the Kernel.**

Our first step is importing the Python libraries that we will use. Select the code below, and execute it using the `Run` button above. Some text should appear to confirm that the PyCh library was imported correctly.

In [1]:

```
# ================================
# Imports
# ================================
from PyCh import *
```

executed in 443ms, finished 14:19:27 2021-11-08

```
PyCh version 1.0 imported succesfully.
```

The next step is to define our KIVA `model`. In our model we define:

- The parameters of our model:
  - Number of robots `N`
  - Rate for storing and retrieving racks `la`
  - Rate for picking racks `mu`
- The simulation environment of our model `env`
- The channels through which processes can communicate: `a`, `b` and `c`
- The processes which live in the simulation environment:
  - Generator process `G`
  - A list of multiple Storage processes `Ss`
  - Buffer process `B`
  - Pick process `P`

The final line of code in the model is that the environment `env` is simulated using `env.run`.

Again, select the code below, and click the `Run` button.

In [2]:

```
# ================================
# KIVA Model
# ================================
def model(N):
    la = 4.0    # average rate / hour for storing and retrieving racks
    mu = 20.0   # average rate / hour for picking of racks
    env = Environment()
    a = Channel(env)
    b = Channel(env)
    c = Channel(env)
    G = Generator(env, a, N)
    Ss = [Storage(env, a, b, la) for j in range(N)]
    B = Buffer(env, b, c)
    P = Pick(env, c, a, mu, 10000)
    env.run()

    throughput = P.value
    print(f"--- The throughput of the sytem with {N} robots is = {throughput:.2f} r
    return throughput

print("--- The model was defined, but it was not yet executed ---")
```

executed in 54ms, finished 14:19:27 2021-11-08

```
--- The model was defined, but it was not yet executed ---
```

As you can see, nothing happened yet. We only defined our model, but we did not yet execute it! However, we first need to define the *process functions* of the `Generator`, `Storage` `Buffer` and `Pick` processes, and the `Pod` entities flowing through the system. Run the code below to define the various processes. Do not try to fully understand the code, we will get back to that in the later chapters!

In [3]:

```
# ================================
# Pod definition
# ================================
@dataclass
class Pod:
    id: int = 0

# ================================
# Generator definition
# ================================
@process
def Generator(env, c_out, N):
    for i in range(N):
        x = Pod()
        send = c_out.send(x)
        yield env.execute(send)

# ================================
# Storage definition
# ================================
@process
def Storage(env, c_in, c_out, la):
    while True:
        receive = c_in.receive()
        x = yield env.execute(receive)

        delay = random.exponential(1.0 / la)
        yield env.timeout(delay)

        send = c_out.send(x)
        yield env.execute(send)

# ================================
# Buffer definition
# ================================
@process
def Buffer(env, c_in, c_out):
    xs = [] # list of pods
    while True:
        receiving = c_in.receive()
        sending = c_out.send(xs[0]) if len(xs)>0 else None
        yield env.select(sending, receiving)
        if selected(sending):
            xs = xs[1:]
        if selected(receiving):
            x = receiving.entity
            xs = xs + [x]


# ================================
# Pick definition
# ================================
@process
```

```
def Pick(env, c_in, c_out, mu, n):
    for i in range(n):
        receive = c_in.receive()
        x = yield env.execute(receive)

        delay = random.exponential(1.0 / mu)
        yield env.timeout(delay)

        send = c_out.send(x)
        yield env.execute(send)
    throughput = n / env.now
    return throughput
```

executed in 9ms, finished 14:19:27 2021-11-08

You are now ready to run your first model! Run the model below to see what the throughput is when the system has `1` robot.

In [4]:

```
model(N = 1);
```

executed in 2.46s, finished 14:19:30 2021-11-08

```
--- The throughput of the sytem with 1 robots is = 3.29 racks/hour---
```

We can even use this model to create an experiment in which we try simulate our system for different numbers of robots. Run the two code cells below.

In [5]:

```
def experiment(low, high):
    throughputData = []
    nRobotsData = []
    for n in range(low, high+1):
        throughput = model(n)
        throughputData = throughputData + [throughput]
        nRobotsData = nRobotsData + [n]
    return throughputData, nRobotsData
```

executed in 4ms, finished 14:19:30 2021-11-08

In [6]:

```
throughputData, nRobotsData = experiment(1,10)
```

executed in 24.5s, finished 14:19:54 2021-11-08

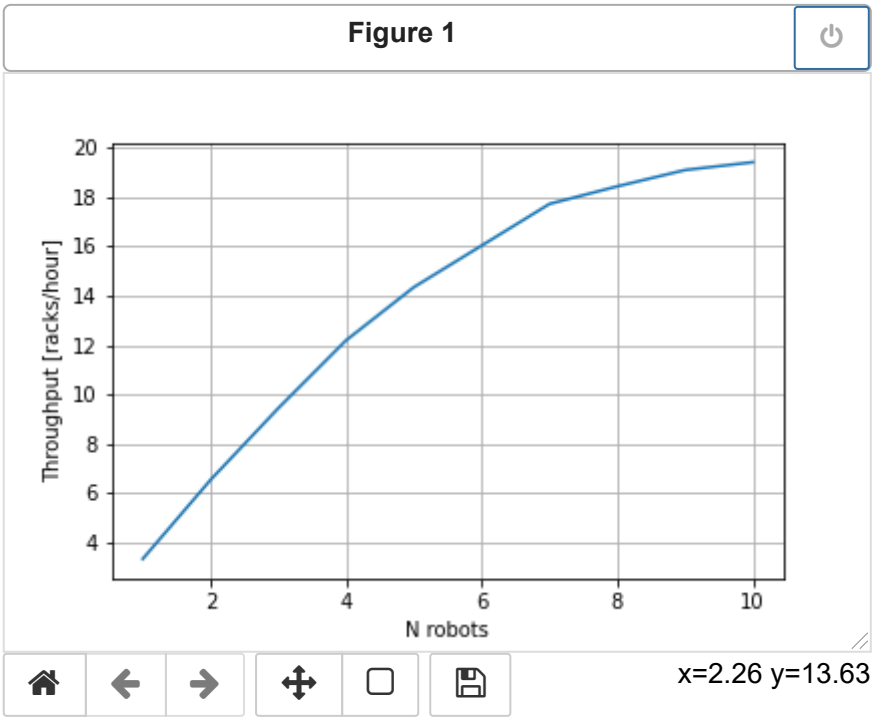
```
--- The throughput of the sytem with 1 robots is = 3.33 racks/hour---
--- The throughput of the sytem with 2 robots is = 6.53 racks/hour---
--- The throughput of the sytem with 3 robots is = 9.44 racks/hour---
--- The throughput of the sytem with 4 robots is = 12.19 racks/hour---
--- The throughput of the sytem with 5 robots is = 14.35 racks/hour---
--- The throughput of the sytem with 6 robots is = 16.03 racks/hour---
--- The throughput of the sytem with 7 robots is = 17.72 racks/hour---
--- The throughput of the sytem with 8 robots is = 18.43 racks/hour---
--- The throughput of the sytem with 9 robots is = 19.10 racks/hour---
--- The throughput of the sytem with 10 robots is = 19.41 racks/hour---
```

We can then use the data gathered in our experiment to create a graph which plots the number of robots against the throughput in racks/hour.

In [7]:

```
fig, ax1 = plt.subplots()
plt.plot(nRobotsData, throughputData, label='sin(x)')
plt.xlabel("N robots")
plt.ylabel("Throughput [racks/hour]")
plt.grid()
```

executed in 36ms, finished 14:19:54 2021-11-08



You can now continue with the rest of the chapters, in which you will learn to model systems such as the KIVA system yourself!

2 Data types

The Python language is a dynamically typed language, which means that you do not have to specify types every time. Still, all variables must be declared in the program. For the declaration of a variable, the data type does not need to be specified, altough this is still possible. Unless specified otherwise, variables with whole number values are integers, while variables that have a decimal point in the value are floating variables.

The fragment below shows the declaration of two elementary data types, integer variable `i` and floating variables `r` and `x`.

In [1]:

```
i = 20
r = 20.0
x = float(20)

print("The data type of i is", type(i), "and its value is", i)
print("The data type of r is", type(r), "and its value is", r)
print("The data type of x is", type(x), "and its value is", x)
```

executed in 5ms, finished 14:19:36 2021-11-08

```
The data type of i is <class 'int'> and its value is 20
The data type of r is <class 'float'> and its value is 20.0
The data type of x is <class 'float'> and its value is 20.0
```

The `print` function in the above code is used to print in the notebook, but more on that in Chapter 5.

The syntax for the declaration of variables is similar to the MATLAB language. An expression, consisting of operators, e.g. plus (`+`), times (`*`), and operands, e.g. `i` and `r`, is used to calculate a new value. The new value can be assigned to a variable by using an assignment statement. An example with four variables, two expressions and assignment statements is:

In [2]:

```
i = 2
r = 1.50
j = 2*i+1
s = r/2

i, r, j, s
```

executed in 10ms, finished 14:19:36 2021-11-08

Out[2]:

```
(2, 1.5, 5, 0.75)
```

The value of variable `j` becomes 5, and the value of `s` becomes 0.75. Assign statements are further described in Chapter 3.

Data types are for now categorized in two different groups: *elementary* types and *container* types. Elementary types are types such as Boolean, integer, float or string. Variables with a container type (a list, set, or dictionary) contain many elements, where each element is of the same type.

2.1 Elementary types

The elementary data types are Booleans, numbers and strings. The language provides the following elementary data types:

- `bool`
- `int`
- `float`
- `string`

2.1.1 Booleans

A boolean value has two possible values, the truth values. These truth values are `False` and `True`. The value `False` means that a property is not fulfilled. A value `True` means the presence of a property. In mathematics, various symbols are used for unary and binary boolean operators. These operators are also present in Python. The most commonly used boolean operators are `not`, `and`, and `or`. The names of the operators, the symbols in mathematics and the symbols in the language are presented in the table below.

Operator	Math	Python
boolean not	¬	`not`
boolean and	∧	`and`
boolean or	∨	`or`

Examples of boolean expressions are the following. If `z` equals `True`, then the value of (`not z`) equals `False`. If `s` equals `False`, and `t` equals `True`, then the value of the expression (`s or t`) becomes `True`. The result of the unary `not`, the binary `and` and `or` operators, for two variables `p` and `q` is given in the table below:

p	q	not p	p and q	p or q
False	False	True	False	False
False	True		False	True
True	False	False	False	True
True	True		True	True

If `p = true` and `q = false`, we find for `p or q` the value `True`.

2.1.2 Numbers

In the language, three types of numbers are available: integer numbers floating numbers and complex numbers, of which only the integer number and floating numbers will be used. Integer numbers are whole numbers, denoted by type int e.g. 3, -10, 0. Float numbers are used to present numbers with a fraction, denoted by type float. E.g. 3.14, 2.7e6 (the scientific notation for 2.7 million). Note that floating numbers must either have a fraction or use the scientific notation, to let the computer know you mean a floating number (in- stead of an integer number). For numbers, the normal arithmetic operators are defined. Expressions can be constructed with these operators. The arithmetic operators are in the table below

Operator name	Notation
raising to the power	`x ** y`
modulo	`x % y`
multiplication	`x * y`
division	`x / y`
unary plus	`+ x`
unary minus	`- x`
addition	`x + y`
subtraction	`x - y`

The priority of the operators is given from high to low. The raising to the power operator has the strongest binding, and the `+` and `-` the weakest binding. E.g., `-3^2` is read as `-(3^2)` and not `(-3)^2`, because the priority rules say that the raising to the power operator binds stronger than the unary operator. Binding in expressions can be changed by the use of parentheses. The integer remainder, denoted by `%`, gives the remainder after division `x / y`. So, `7 % 3` gives `1` and `-7 % 3` gives `2`, `7 % 3`. The rule for the result of an operation is as follows. If one of the operands is of type `float`, the result of the operation is of type `float`. Conversion functions exist to convert a `float` into an integer. The function `ceil` converts a float to the smallest integer value not less than the float, the function `floor` gives the biggest integer value smaller than or equal to the float, and the function `round` rounds the float to the nearest integer value (or up, if it ends on `0.5`). Note that for the functions `ceil` and `floor`, the "math" library has to be imported by entering `import math`, e.g. at the beginning of the notebook. Between two numbers, a relational operation can be defined. If e.g. variable `x` is smaller than variable `y`, the expression `x < y` equals true. The relational operators, with well known semantics, are listed below:

Name	Operator
less than	`x < y`
at most	`x <= y`
equals	`x == y`

Name	Operator
differs from	`x != y`
at least	`x >= y`
greater than	`x > y`

2.1.3 Strings

Variables of type string contains a sequence of characters. A string is enclosed by single or double quotes. An example is `"Manufacturing networks"`. Strings can be composed from different strings. The concatenation operator (`+`) adds one string to another, e.g. `"Systems" + " " + "engineering"` gives `"Systems engineering"`. Moreover the relational operators (`<`, `<=`, `==`, `!=`, `>=`, and `>`) can be used to compare strings alphabetically, e.g. `"a" < "aa" < "ab" < "b"`.

2.2 Container types

Lists, *sets* and *dictionaries* are container types. A variable of this type contains zero or more identical elements. Elements can be added or removed in variables of these types.

Sets are unordered collections of elements. Each element value either exists in a set, or it does not exist in a set. Each element value is unique, duplicate elements are silently discarded. A *list* is an ordered collection of elements, i.e. there is a first and a last element (in a non-empty list). A list also allows duplicate element values. *Dictionaries* are unordered and have no duplicate value, just like sets, but you can associate a value (of a different type) with each element value.

Lists are denoted by a pair of (square) brackets. For example, `[7, 8, 3]` is a list with three integer elements. Since a list is ordered, `[8, 7, 3]` is a different list.

Sets are denoted by a pair of (curly) braces, e.g. `{7, 8, 3}` is a set with three integer elements. A set is an unordered collection of elements. The set `{7, 8, 3}` is a set with three integer numbers. Since order of the elements does not matter, the same set can also be written as `{8, 3, 7}` (or in one of the four other orders). In addition, each element in a set is unique, e.g. `{8, 7, 8, 3}` is equal to `{7, 8, 3}`. For readability, elements in a set are normally written in increasing order, i.e. as `{3, 7, 8}`.

Dictionaries are denoted by a pair of (curly) braces, whereby an element value consists of two parts, a "key" and a "value" part. The two parts separated by a colon (`:`). For example `{"jim" : 32, "john" : 34}` is a dictionary with two elements. The first element has `"jim"` as key part and `32` as value part, the second element has `"john"` as key part and `34` as value part. The key parts of the elements work like a set, they are unordered and duplicates are silently discarded. A value part is associated with its key part. In this example, the key part is the name of a person, while the value part keeps the age of that person.

Container types have some built-in functions (Functions are described in Chapter 4) in common:

- The function `len` gives the number of elements in a variable. E.g. `len([7, 8, 3])` yields `3`; `len({7, 8})` results in `2`; `len({"jim":32})` gives `1` (an element consists of two parts).

- To check whether there are no elements in a variable, in other words, to chekc whether a variable is empty, the boolean operator `not` can be used. E.g., `not []` returns `True`, and `not [123]` yields `False`
- The function `[variable].pop()` extracts a value from the provided collection and returns that extracted value, and the collection is updated without that value. For lists, the index of the value that should be extracted can be defined. Note that unlike MATLAB, the first index of a list is indicated by 0! For sets, only the first item can be extracted. For dictionaries, an item can only be discarded when the "key"-part is used in the pop-function. What will be returned is the "value"-part of that item in the dictionary. An example is shown below:

In [3]:

```
x = [7, 1, 0, 3]
q = x.pop(3)
print('The new x = ',x)
print('The extracted value from x is', q)

y = {7, 1, 0, 3}
r = y.pop()
print('The new y = ',y)
print('The extracted value from y is', r)

z = {"jim":32,"john":34}
s = z.pop("john")
print('The new z = ',z)
print('The extracted value from z is', s)
```

executed in 6ms, finished 14:19:37 2021-11-08

```
The new x =  [7, 1, 0]
The extracted value from x is 3
The new y =  {1, 3, 7}
The extracted value from y is 0
The new z =  {'jim': 32}
The extracted value from z is 34
```

2.2.1 Lists

A list is an ordered collection of elements of the same type. They are useful to model anything where duplicate values may occur or where order of the values is significant. E.g. waiting customers in a shop, process steps in a recipe, or products stored in a warehouse. Various operations are defined for lists.

An element can be fetched by *indexing*. This indexing operation does not change the content of the variable. The first element of a list has index `0`. The last element of a list has index `len(xs) - 1`. A negative index, say `m`, starts from the back of the list, or equivalently, at offset `len(xs) + m` from the front. You cannot index non-existing elements. Some examples, with `xs = [7, 8, 3, 5, 9]` are:

In [4]:

```
xs = [7, 8, 3, 5, 9]
print(xs[0]) # -> 7
print(xs[3]) # -> 5
```

executed in 3ms, finished 14:19:37 2021-11-08

```
7
5
```

In [5]:

```
print(xs[5]) # -> ERROR (there is no element at position 5)
```

executed in 67ms, finished 14:19:37 2021-11-08

```
---------------------------------------------------------------------------
----
IndexError                                Traceback (most recent call l
ast)
<ipython-input-5-aa5d54d620a8> in <module>
----> 1 print(xs[5]) # -> ERROR (there is no element at position 5)

IndexError: list index out of range
```

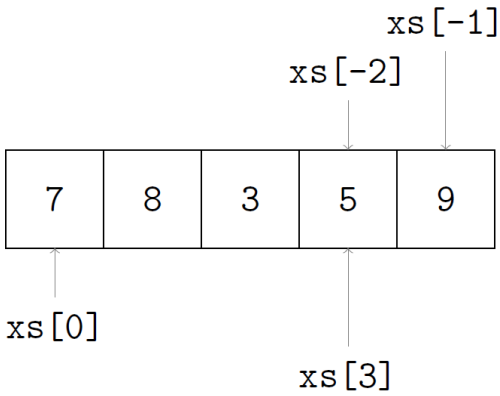
In [ ]:

```
print(xs[-1]) # -> xs[5 - 1] -> xs[4] -> 9
print(xs[-2]) # -> xs[5 - 2] -> xs[3] -> 5
```

executed in 100ms, finished 14:19:37 2021-11-08

In Figure 2.1, the list with indices is visualized.

Figure 2.1: A list with indices



A part of a list can be fetched by *slicing*. The slicing operation does not change the content of the list, it copies a contiguous sequence of a list. The result of a slice operation is again a list, even if the slice contains just one element. Slicing is denoted by `xs[i:j]`. The slice of `xs[i:j]` is defined as the sequence of elements with index `k` such that `i <= k < j`. Note the upper bound `j` is noninclusive. If

`i` is omitted use `0`. If `j` is omitted use `len(xs)`. If `i` is greater than or equal to `j`, the slice is empty. If `i` or `j` is negative, the index is relative to the end of the list: `len(xs) + i` or `len(xs) + j` is substituted. Some examples with `xs = [7, 8, 3, 5, 9]`:

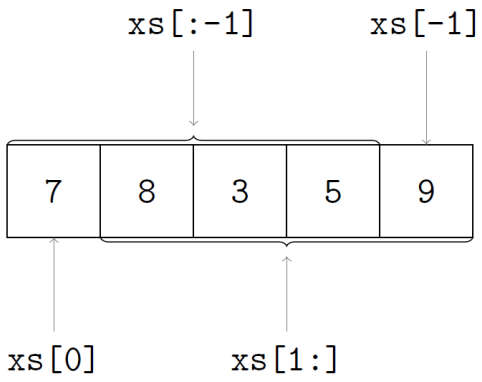
In [ ]:

```
xs = [7, 8, 3, 5, 9]
print(xs[1:3]) # -> [8, 3]
print(xs[:2]) # -> [7, 8]
print(xs[1:]) # -> [8, 3, 5, 9]
print(xs[:-1]) # -> [7, 8, 3, 5]
print(xs[:-3]) # -> [7, 8]
```

executed in 74ms, finished 14:19:37 2021-11-08

A common name for the first element of a list (i.e., `x[0]`) is the head of a list. The list of all but the first elements (`xs[1:]`) is often called tail. Similarly, the last element of a list (`xs[-1]`) is also known as head right, and `xs[:-1]` is also known as tail right. In Figure 2.2 the slicing operator is visualized.

Figure 2.2: A list with indices and slices



Two lists can be "glued" into a new list. The glueing or concatenation of a list with elements `7, 8, 3` and a list with elements `5,` and `9` is de noted by:

In [ ]:

```
print([7, 8, 3] + [5, 9]) # -> [7, 8, 3, 5, 9]
```

executed in 48ms, finished 14:19:37 2021-11-08

An element can be added to a list at the rear or at the front. The action is performed by transforming the element into a list and then concatenate these two lists. In the next example the value `5` is added to the rear, respectively the front, of a list:

In [6]:

```
print([7, 8, 3] + [5]) # -> [7, 8, 3, 5]
print([5] + [7, 8, 3]) # -> [5, 7, 8, 3]
```

executed in 4ms, finished 14:19:37 2021-11-08

```
[7, 8, 3, 5]
[5, 7, 8, 3]
```

Lists also have some other, click here (https://www.w3schools.com/python/python_ref_list.asp) for more info.

- The `remove` function removes the first element of a given value, e.g. `xs.remove(4)` returns the list `xs` with the first element of value `4` removed.
- The `append` function adds a single element at the end of the list, e.g. `xs.append(3)` returns the list `xs` with value `3` at the end.
- The `extend` function adds one or more elements at the end of the listt, e.g. `xs.extend([2, 4])` returns the list `xs` with values `2` and `4` at the end.
- The `insert` function adds an element at the specified position, e.g. `xs.insert(3,10)` inserts the value `10` at index `3` (shifting the following elements by 1).
- The `index` function returns the index of the first element with the specified value, e.g. `xs.index(10)` returns the index of the first element with value `10` .

In [7]:

```
xs = [1, 4, 2, 4, 5]
print(f"We start with list {xs}")

xs.remove(4) # ->  [1, 2, 4, 5]
print(f"After remove(4) we end up with list {xs}")

xs.append(3) # -> [1, 2, 4, 5, 3]
print(f"After append(4) we end up with list {xs}")

xs.extend([2, 4]) # -> [1, 2, 4, 5, 3, 2, 4]
print(f"After extend([2, 4]) we end up with list {xs}")

xs.insert(3,10) # -> [1, 2, 4, 10, 5, 3, 2, 4]
print(f"After insert(3,10) we end up with list {xs}")

ind = xs.index(4)
print(f"We find value 4 at index {ind}")
```

executed in 7ms, finished 14:19:37 2021-11-08

```
We start with list [1, 4, 2, 4, 5]
After remove(4) we end up with list [1, 2, 4, 5]
After append(4) we end up with list [1, 2, 4, 5, 3]
After extend([2, 4]) we end up with list [1, 2, 4, 5, 3, 2, 4]
After insert(3,10) we end up with list [1, 2, 4, 10, 5, 3, 2, 4]
We find value 4 at index 2
```

In [8]:

```
xs = [1, 4, 2, 4, 5]
xs.remove(8) # -> ERROR (8 does not occur in the list)
```

executed in 15ms, finished 14:19:37 2021-11-08

```
---------------------------------------------------------------------
----
ValueError                                Traceback (most recent call l
ast)
<ipython-input-8-bff5e801b6c7> in <module>
      1 xs = [1, 4, 2, 4, 5]
----> 2 xs.remove(8) # -> ERROR (8 does not occur in the list)

ValueError: list.remove(x): x not in list
```

Lists have two relational operators, the equal operator and the not-equal operator. The equal operator (`==`) compares two lists. If the lists have the same number of elements and all the elements are pair-wise the same, the result of the operation is `True` , otherwise `False` . The not-equal operator (`!=`) does the same check, but with an opposite result. Some examples, with `xs = [7, 8, 3]` :

In [ ]:

```
xs = [7, 8, 3]
print(xs == [7, 8, 3]) # -> True
print(xs == [7, 7, 7]) # -> False
```

executed in 101ms, finished 14:19:37 2021-11-08

The membership operator (`in`) checks if an element is in a list. Some examples, with `xs = [7, 8, 3]` :

In [ ]:

```
xs = [7, 8, 3]
print(6 in xs) # -> False
print(7 in xs) # -> True
print(8 in xs) # -> True
```

executed in 80ms, finished 14:19:37 2021-11-08

Last, but not least, Python has something called **List comprehension**. List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list. More information on list comprehension can be found here (https://www.w3schools.com/python/python_lists_comprehension.asp). List comprehension follows the following syntax:

```
newlist = [<expression> for <item> in <list> if <condition>]
```

Which translates to: for every *item* in the *list* which adheres to the given *condition*, add *expression* to the *new list*.

Some examples are shown below:

In [ ]:

```
xs = [7, 3, 6, 4, 8, 2, 9, 1]
newlist = [1 for x in xs]
print(newlist)
```

executed in 64ms, finished 14:19:37 2021-11-08

In [ ]:

```
xs = [7, 3, 6, 4, 8, 2, 9, 1]
newlist = [x for x in xs if x < 5]
print(newlist)
```

executed in 61ms, finished 14:19:37 2021-11-08

In [ ]:

```
xs = [7, 3, 6, 4, 8, 2, 9, 1]
newlist = [-x*10 for x in xs if x < 5]
print(newlist)
```

executed in 61ms, finished 14:19:37 2021-11-08

In [ ]:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [fruit for fruit in fruits if "a" in fruit]
print(newlist)
```

executed in 58ms, finished 14:19:37 2021-11-08

2.2.2 Sets

Set operators for union, intersection and difference are present. The table below gives the name, the mathematical notation and the notation in the language. The union of two sets merges the values of both sets into one, that is, the result is the collection of values that appear in at least one of the arguments of the union operation.

Operator	Math	Python
set union	∪	`.union`
set intersection	∩	`.intersection`
set difference	\	`.difference`

Some examples:

In [9]:

```
xs = {3, 7, 8};
print(xs.union({7, 9})) # -> {3, 7, 8, 9}
```

executed in 4ms, finished 14:19:37 2021-11-08

```
{3, 7, 8, 9}
```

All permutations with the elements `3, 5, 7, 8` and `9` are correct (sets have no order, all permutations are equivalent).

Values that occur in both arguments, appear only one time in the result (sets silently discard duplicate elements).

The intersection of two sets gives a set with the common elements, that is, all values that occur in both arguments. Some examples:

In [10]:

```
xs = {3, 7, 8}
print(xs.intersection({5, 9})) # no common element, empty set
print(xs.intersection({7, 9})) # only 7 in common
```

executed in 4ms, finished 14:19:37 2021-11-08

```
set()
{7}
```

Set difference works much like subtraction on lists, except elements occur at most one time (and have no order). The operation computes "remaining elements". The result is a new set containing all values from the first set which are not in the second set. Some examples:

In [11]:

```
xs = {3, 7, 8}
print(xs.difference({5, 9})) # -> {8, 3, 7}
print(xs.difference({7, 9})) # -> {8,3}
```

executed in 5ms, finished 14:19:37 2021-11-08

```
{8, 3, 7}
{8, 3}
```

The membership operator in works on sets too:

In [12]:

```
print(3 in {3, 7, 8}) # -> True
print(9 in {3, 7, 8}) # -> False
```

executed in 4ms, finished 14:19:37 2021-11-08

```
True
False
```

2.2.3 Dictionaries

Elements of dictionaries are stored according to a key, while lists elements are ordered by a (relative) position, and set elements are not ordered at all. A dictionary can grow and shrink by adding or removing elements respectively, like a list or a set. An element of a dictionary is accessed by the key of the element.

The dictionary variable `d` of type `(string : int)` is given by:

In [13]:

```
d = {"jim" : 32,
     "john" : 34,
     "adam" : 25}
print(d)
```

executed in 4ms, finished 14:19:37 2021-11-08

```
{'jim': 32, 'john': 34, 'adam': 25}
```

Retrieving values of the dictionary by using the key:

In [14]:

```
print(d["john"]) # -> 34
print(d["adam"]) # -> 25
```

executed in 4ms, finished 14:19:37 2021-11-08

```
34
25
```

Using a non-existing key to retrieve a value results in a error message.

A new value can be assigned to the variable by selecting the key of the element:

In [15]:

```
d["john"] = 35
print(d)
```

executed in 3ms, finished 14:19:37 2021-11-08

```
{'jim': 32, 'john': 35, 'adam': 25}
```

This assignment changes the value of the `"john"` item to `35`. The assignment can also be used to add new items:

In [16]:

```
d["lisa"] = 19
print(d)
```

executed in 3ms, finished 14:19:37 2021-11-08

```
{'jim': 32, 'john': 35, 'adam': 25, 'lisa': 19}
```

Membership testing of keys in dictionaries can be done with the in operator:

In [17]:

```
print("jim" in d) # -> Rrue
print("peter" in d) # -> False
```

executed in 3ms, finished 14:19:37 2021-11-08

```
True
False
```

Merging two dictionaries is done by adding them together. The value of the second dictionary is used when a key exists in both dictionaries:

In [18]:

```
print( {**{1 : 1, 2 : 2},**{1 : 5, 3 : 3}} ) # -> {1 : 5, 2 : 2, 3 : 3}
```

executed in 3ms, finished 14:19:37 2021-11-08

```
{1: 5, 2: 2, 3: 3}
```

The left dictionary is copied, and updated with each item of the right dictionary.

Removing elements can be done with subtraction, based on key values. Lists and sets can also be used to denote which keys should be removed. A few examples for `d` is `{1 : 1, 8 : 2}`:

In [19]:

```
d = {1 : 1, 8 : 2}
d.pop(8)
print(d)
```

executed in 5ms, finished 14:19:40 2021-11-08

```
{1: 1}
```

Subtracting keys that do not exist in the left dictionary is not allowed.

2.3 Exercises

Exercise 2.3.1

Exercises for integer numbers. What is the result of the following expressions:

```
-5 ** 3
-5 * 3
-5 % 3
```

In [ ]:

```
```

Exercise 2.3.2

Exercises for lists. Given is the list `xs = [0,1,2,3,4,5,6]` .
Determine the outcomes of:

```
xs[0]
xs[1:]
len(xs)
xs + [3]
[4,5] + xs
xs.remove(2)
xs[0] + (xs[1:])[0]
```

In [ ]:

```
```

Exercise 2.3.3

Exercises for list comprehension. Given is the list `xs = [0,8,5,3,7,6,2]`
Create the following lists using list comprehension:

- All elements of xs, with their values multiplied by 3
- All elements of xs lower than 7 and higher than 2
- All elements of xs higher than 3, with their values divided by 3

In [ ]:

```
```

2.4 Answers to exercises

Answer to 2.3.1

[Click for the answer to 2.3.1]

Answer to 2.3.2

[Click for the answer to 2.3.2]

Answer to 2.3.3

[Click for the answer to 2.3.3]

3 Statements

There are several kinds of statements, such as assignment statements, choice statements (select and if statements), and loop statements (while and for statements). Semicolons are **not** required after statements, however, a semicolon can surpress the output of the last statement.

3.1 The assignment statement

An *assignment* statement is used to assign values to variables. Two examples:

In [1]:

```
x = 2
y = x + 10

print(x, y)
```

executed in 4ms, finished 14:19:48 2021-11-08

```
2 12
```

The first assignment, assigns the integer value `2` to `x`. It concists of the name of the variable (`x`), an assignment symbol (`=`), and value `2`. The second assignment consists of a name of the variable (`y`), an assignment symbol (`=`), and an expression (`x + 10`) yielding a value. When `x` is `2`, the value of the expression for `y` is `12`. Execution of this statement copies the value to the `y` variable, immediately after executing the assignment, the value of the `y` variable is `10` larger than the value of the `x` variable at this point of the program.

The value of the `y` variable will not change until the next assignment to `y`, for example, performing the assignment `x = x + 7` has no effect on the value of the `y` variable. This is shown below:

In [2]:

```
x = x + 7
print(x, y)
```

executed in 3ms, finished 14:19:48 2021-11-08

```
9 12
```

Independent assignments can also be combined in a multi-assignment, e.g.:

In [3]:

```
i, j = x+1, 1
print(i, j)
```

executed in 3ms, finished 14:19:48 2021-11-08

```
10 1
```

The result is the same as the above described example, the first value goes into the first variable, the second value into the second variable, etc. In an assignment statement, first all expression values are computed before any assignment is actually done. In the following example the values of `x` and `y` are swapped:

In [4]:

```
x = 1
y = 2
x, y = y, x
print(x, y)
```

executed in 3ms, finished 14:19:48 2021-11-08

```
2 1
```

3.2 The if statement

The *if* statement is used to express decisions. In Python there are no `end` statements; instead indentations are used to indicate the scope of the statement block. An example:

In [5]:

```
x = -1
if x < 0:
    y = -x  # indented, and thus in the scope of the if statement.

print(x, y)
```

executed in 3ms, finished 14:19:48 2021-11-08

```
-1 1
```

If the value of `x` is negative, assign its negated value to `y`. Otherwise, do nothing (skip the `y = -x` assignment statement).

To perform a different statement when the decision fails, an `if`-statement with an else alternative can be used. It has the following form. An example:

In [6]:

```
a = 10
b = 4
if a > 0:
    c = a
else:
    c = b

print(c)
```

executed in 3ms, finished 14:19:48 2021-11-08

```
10
```

If `a` is positive, variable `c` gets the value of `a`, otherwise it gets the value of `b`. Try it out with different values of a and b!

In some cases more alternatives must be tested. One way of writing it is by nesting an `if`-statement in the `else` alternative of the previous `if`-statement.

In [7]:

```
i = 12
if i < 0:
    print("i < 0")
else:
    if i == 0:
        print("i = 0")
    else:
        if i > 0 and i < 10:
            print("0 < i < 10")
        else:
            # i must be greater or equal 10
            print("i >= 10")
```

executed in 3ms, finished 14:19:48 2021-11-08

```
i >= 10
```

This tests `i < 0`. If it fails, the `else` is chosen, which contains a second `if`-statement with the `i == 0` test. If that test also fails, the third condition `i > 0` and `i < 10` is tested, and one of the `print`-statements is chosen. The above can be written more compactly by combining an `else`-part and the `if`-statement that follows, into an `elif` part. Each `elif` part consists of a boolean expression, and a statement list. Using `elif` parts results in:

In [8]:

```
i = 12
if i < 0:
    print("i < 0")
elif i == 0:
    print("i = 0")
elif i > 0 and i < 10:
    print("0 < i < 10")
else:
    # i must be greater or equal 10
    print("i >= 10")
```

executed in 4ms, finished 14:19:48 2021-11-08

```
i >= 10
```

Each alternative starts at the same column, instead of having increasing indentation. The execution of this combined statement is still the same, an alternative is only tested when the conditions of all previous alternatives fail.

Note that the line " `# i must be greater or equal 10` " is a comment to clarify when the alternative is chosen. It is not executed by the simulator. You can write comments either at a line by itself like above, or behind program code. It is often useful to clarify the meaning of variables, give a more detailed explanation of parameters, or add a line of text describing what the purpose of a block of code is from a birds-eye view.

3.3 The while statement

The *while* statement is used for repetitive execution of the same statements, a so-called *loop*. A fragment that calculates the sum of `10` integers, `10` , `9` , `8` , `...` , `3` , `2` , `1` , is:

In [9]:

```
i = 10
Sum=0
while i > 0:
    Sum = Sum + i; i = i - 1

print(Sum)
```

executed in 4ms, finished 14:19:48 2021-11-08

```
55
```

Each iteration of a `while` statement starts with evaluating its condition (`i > 0` above). When it holds, the statements inside the while (the `Sum = Sum + i; i = i - 1` assignments) are executed (which adds `i` to the sum and decrements `i`). At the end of the statements, the `while` is executed again by evaluating the condition again. If it still holds, the next iteration of the loop starts by executing the assignment statements again, etc. When the condition fails (`i` is equal to `0`), the `while` statement ends, and execution continues beyond the `while` code block.

A fragment with an infinite loop is:

In [*]:

```
i = 0
while True:
    i = i + 1;
    # This code can be terminated by pressing the stop-sign in the menu at the
    # top
```

execution queued 14:19:48 2021-11-08

The condition in this fragments always holds, resulting in `i` getting incremented "forever". Such loops are very useful to model things you switch on but never off, e.g. processes in a factory.

A fragment to calculate $z = x^y$, showing the use of two while loops, is:

In [*]:

```
x = 2
y = 3
z = 1
while y > 0:
    while y % 2 == 0:
        y = y / 2; x = x * x
    y = y - 1; z = x * z
print(z)
```

execution queued 14:19:48 2021-11-08

A fragment to calculate the greatest common divisor (GCD) of two integer numbers `j` and `k`, showing the use of `if` and `while` statements, is:

In [*]:

```
j = 28
k = 77
while j != k:
    if j > k:
        j = j - k
    else:
        k = k - j
print(k)
```

execution queued 14:19:48 2021-11-08

The symbol `!=` stands for "differs from" ("not equal").

3.4 The for statement

The `while` statement is useful for looping until a condition fails. The *for* statement is used for iterating over a collection of values. A fragment with the calculation of the sum of `10` integers:

In [*]:

```
Sum = 0
for i in range(1,11):
    Sum = Sum + i
print(Sum)
```

execution queued 14:19:48 2021-11-08

The result of the expression `range(1, 11)` is a list whose items are consecutive integers from `1` (included) up to `11` (excluded): `[1, 2, 3, ..., 9, 10]`. The following example illustrates the use of the `for` statement in relation with container-type variables. Another way of calculating the sum of a list of integer numbers:

In [*]:

```
xs = [1,2,3,5,7,11,13]
Sum = 0
for x in xs:
    Sum = Sum + x
print(Sum)
```

execution queued 14:19:48 2021-11-08

This statement iterates over the elements of list `xs`. This is particularly useful when the value of `xs` may change before the `for` statement.

3.5 Notes

In this chapter the most used statements are described. The language offers the following extensions:

1. Inside loop statements *break* and *continue* statements are allowed. The `break` statements allows "breaking out of a loop", that is, abort a `while` or a `for` statement. The `continue` statement aborts execution of the statements in a loop. It "jumps" to the start of the next iteration.
2. A rarely used statement is the *pass* statement. It's like an `x = x` assignment statement, but more clearly expresses "nothing is done here".

3.6 Exercises

Exercise 3.6.1

Study the specification below and explain why, though it works, it is not an elegant way of modelling the selection. Make a suggestion for a shorter, more elegant version.

In [ ]:

```
# Original code:
i = 3
if (i < 0) == True:
    print(i,"is a negative number")
elif (i <= 0) == False:
    print(i,"is a positive number")
```

In [ ]:

```
# More elegant version:
```

Exercise 3.6.2

Construct a list with the squares of the first 10 integers:

- **(a)** using a `for` statement.
- **(b)** using a `while` statement.

In [ ]:

```
# 2a: Using for statement
```

In [ ]:

```
# 2b: Using while statement
```

Exercise 3.6.3

Write a program that:

- **(a)** makes a list with the first 50 prime numbers.
- **(b)** Extend the program with computing the sum of the first 7 prime numbers.
- **(c)** Extend the program with computing the sum of the last 11 prime numbers.

In [ ]:

```
# 3a: generate a list of 50 prime numbers

primes = []
#...
```

In [ ]:

```
# 3b: Extend the program with computing the sum of the first 7 prime numbers.
# Using the previously calculated list of primes in this answer (no need to recalcula
```

In [ ]:

```
# 3c: Extend the program with computing the sum of the last 11 prime numbers.
# Again, use the previously calculated list of primes in this answer (no need to re
```

3.7 Answers to exercises

Answer to 3.6.1

[Click for the answer to 3.6.1]

Answer to 3.6.2

[Click for the answer to 3.6.2]

Answer to 3.6.3

[Click for the answer to 3.6.3]

To use the examples in this chapter, first run the code below to import the right libraries.

In [1]:

```
# ================================
# Imports
# ================================
import math
from dataclasses import dataclass
```

executed in 5ms, finished 14:21:56 2021-11-08

4 Functions and classes

Within Python, functions and classes can be used to organize a script, and to reuse certain parts of a script easily.

4.1 Function definitions

In a model, computations must be performed to process the information that is sent around. Short and simple calculations are written as assignments between the other statements, but for longer computations or computations that are needed at several places in the model, a more encapsulated environment is useful, a *function*. In addition, the language comes with a number of built-in functions, such as `len` or `pop` on container types (possibly by importing libraries). An example:

In [2]:

```
def mean(xs):
    Sum = 0
    for x in xs:
        Sum = Sum + x
    return Sum / len(xs)
```

executed in 3ms, finished 14:21:56 2021-11-08

The `def` keyword indicates that a function is defined. It is followed by the name of the function, in this example " `mean` ". Between the parentheses, the function's arguments (the *input parameters*) are listed. In this example, there is one argument named `xs` . Parameter `xs` is then refered to in the body of the function.

The colon `:` at the end of the first line indicates the start of the computation. Below it are new variable declarations (`Sum = 0`), and statements to compute the value, the *function algorithm*. The `return` statement denotes the end of the function algorithm. The value of the expression behind it the function's output (the result of the calculation). This example computes and returns the mean value of the integers of the list.

Use of a function (*application* of a function) is done by using its name, followed by the values to be used as input (the *actual parameters*). How the above function can be used is shown below.

The actual parameter of this function application is `[1, 3, 5, 7, 9]` . The function result is `(1 + 3 + 5 + 7 + 9)/5` (which is `5.0`).

In [3]:

```
m = mean([1, 3, 5, 7, 9])
print(m)
```

executed in 3ms, finished 14:21:56 2021-11-08

```
5.0
```

A function is a mathematical function: the result of a function depends on the input parameters. However, a function has access to global variables (any variable which is defined outside a function). An example is shown below, where the `record` function is used to add variables to the `data` list.

The record function does not have a return statement, which means that the function will end (without an output) once all its steps have finished executing.

In [4]:

```
data = []

def record(x):
    data.append(x)

record(1)
record(3)
record(-5)

print(data)
```

executed in 3ms, finished 14:21:56 2021-11-08

```
[1, 3, -5]
```

Note: it is very easy to make mistakes when using global variables, so do not use this if you are not sure what you are doing! It is very easy to lose track of which functions make changes to a global variable. It is better to pass all variables you want a function to use as parameters.

It is possible to use multiple `return` statements in a function. An example is shown in the function below, which calculates the sign of a real number. The sign function returns: if `r` is smaller than zero, the value minus one; if `r` equals zero, the value zero; and if `r` is greater than zero, the value one. The computation in a function ends when it encounters a `return` statement. The `return 1` at the end is therefore only executed when both `if` conditions are false.

In [5]:

```
def sign(r):
    if r < 0:
        return -1
    elif r == 0:
        return 0
    return 1
```

executed in 3ms, finished 14:21:56 2021-11-08

In [6]:

```
sign(-5)
```

executed in 11ms, finished 14:21:56 2021-11-08

Out[6]:

```
-1
```

4.2 Dataclasses

Python classes can help to create structures and functionalities in a code, so that the code becomes easier to maintain, and reuse.

Classes can be used to create a structure in which data can be stored, together with functionalities that can be defined manually. Creating a new class creates a new type of object, of which new instances can be made. Each class instance can have attributes, which could also be modified afterwards. In this course, we will focus on one specific type of class: a dataclass. Dataclasses are especially useful when modeling entities moving through a production line. We will not go into detail on classes in general, but if you want to learn more about them you can click here (https://docs.python.org/3/tutorial/classes.html). More on dataclasses can be found here (https://docs.python.org/3/library/dataclasses.html). To make use of dataclasses, firstly enter `from dataclasses import dataclass` .

An example of a dataclass is shown below:

In [7]:

```
@dataclass
class Person:
    # Class for keeping track of an item in inventory.
    name: str
    age: int
    height: float

person1 = Person("Mark", 45, 1.75)

print(person1.name)
print(person1.height)

person1.name = "Tom"
print(person1.name)
```

executed in 5ms, finished 14:21:56 2021-11-08

```
Mark
1.75
Tom
```

In the above example, the new dataclass `Person` is defined, which has the properties `name`, `age`, and `height`, with the data types `str`, `int`, and `float` respectively. A dataclass requires you to explicitly define the datatypes of its properties (which is different from defining normal variables). The expression `Person("Mark", 45, 1.75)` assigns values to the properties of the dataclass `Person`. Note that the expression `Person()` requires exactly three inputs.

It is also possible to define a default value for a property in a dataclass, this is shown in the example below:

In [8]:

```
@dataclass
class Product:
    # Class for keeping track of a product.
    producttype: str
    production_time: float
    quantity: int = 1

q = Product("Banana",12.5)
print(q.quantity)

r = Product("Banana",12.5,6)
print(r.quantity)
```

executed in 4ms, finished 14:21:56 2021-11-08

```
1
6
```

In this case, the default value of the property `quantity` of the dataclass `Product` is 1. When defining

a default property in a dataclass, it should be defined **after** the properties that do not have a default value. Furthermore, when initiating such a dataclass - in this case `q = Product("Banana",0.25)` - only two inputs are required, however, the property `quantity` can still be defined by adding another input.

Functions can be defined within a dataclass as well. An example:

In [9]:

```
@dataclass
class Costumer:
    name: str
    questions: int

    def answer(self):
        if self.questions > 0:
            return self.questions - 1

C1 = Costumer("John",5)
C1.questions = Costumer.answer(C1)
print(C1.questions)
```

executed in 9ms, finished 14:21:56 2021-11-08

```
4
```

In this case, the dataclass `Costumer` has two properties: `name` and `questions`. Furhtermore, the function `answer` is defined. This function calculates the subtraction of the property `questions` with one. Such a function can be called using the dataclass and function as follows: `Costumer.answer(C1)`, where `C1` is an instance of the dataclass `Costumer`. In the example above, the property `questions` is redefined using the `answer` function.

4.3 Exercises

Exercise 4.3.1

Derive a fibonacci-function which outputs the n-th number of the Fibonacci sequence. So, `fibonacci(0) = 0`, `fibonacci(1) = 1`, `fibonacci(2) = 1`, and `fibonacci(21) = 10946`.

In [ ]:

```
```

Exercise 4.3.2

Write the following code in a more elegant way, such that the function `total_production_time` can only be used for a specific dataclass. Do this by defining a dataclass, including a function defined for the dataclass only.

In [ ]:

```
# The original code without dataclasses

def total_production_time(amount,production_time):
    return amount * production_time

# Car 1
Car1_amount = 4000
Car1_production_time = 30

# Car 2
Car2_amount = 5000
Car2_production_time = 25

# Car 3
Car3_amount = 24000
Car3_production_time = 9

print(total_production_time(Car1_amount, Car1_production_time))
print(total_production_time(Car2_amount, Car2_production_time))
print(total_production_time(Car3_amount, Car3_production_time))
```

In [ ]:

```
# Your code with dataclasses.

```

4.4 Answers to exercises

Answer to 4.3.1

[Click for the answer to 4.3.1]

Answer to 4.3.2

[Click for the answer to 4.3.2]

To use the examples in this chapter, first run the code below to import the right libraries.

We use the numpy library show some examples. However, you do not need to learn how to use this library (except the `random` module, but more on that in next chapter).

In [1]:

```
# ================================
# Imports
# ================================
from numpy import random
from dataclasses import dataclass
import math
import numpy as np
import matplotlib.pyplot as plt
```

executed in 216ms, finished 14:22:04 2021-11-08

5 Input and output

This chapter describes how data can be read from a file as input for a model, and how the output of a model can be printed to the screen, written to a file, or plotted in a graph.

5.1 Print

The `print` statement is used for output of data to the screen of the computer. A simple example is shown below.

In [2]:

```
x = 5
print("x =", x)
```

executed in 3ms, finished 14:22:04 2021-11-08

```
x = 5
```

The above example is very useful for the most simple use cases; we've used similar print statements throughout the previous chapters. However, it is not very versatile. For example, try changing `x = 5` to `x= 5/3` . The result is that `x` is printed with many decimals.

In the next example we will use an f-string, which allows us to insert variables into strings (there are more ways for doing so in Python, for more information on f-string and other options click here (https://realpython.com/python-f-strings/#f-strings-a-new-and-improved-way-to-format-strings-in-python)). An f-string by placing a `f` in front of the string, two examples are shown below.

In [3]:

```
i = 5/3

print("i =", i) # the basic method

print(f"i = {i}") # Using f-strings to insert variable i into the string

print(f"i = {i:.2f}") # Using f-strings to insert variable i into the string (with
```

executed in 4ms, finished 14:22:04 2021-11-08

```
i = 1.6666666666666667
i = 1.6666666666666667
i = 1.67
```

The above example shows multiple ways to print `i = 5/3` to the screen. Let us take the last example: `f"i = {i:.2f}"` . The `f` before the string indicates that the string is formatted as an f-string. The curly brackets indicate that we want to insert a variable, and the formatting we want to use in the form of `{variable:format}` (or `{variable}` if we do not wish to specify the format). In this case the variable is `i` and format is `.2f` (which denotes *fixed-point number format* with 2 decimal numbers). There are many different format types, such as `d` (for integer numbers), `f` (for floats), and `g` (for any number).

Another example:

In [4]:

```
i = 5
r = 3.14
print(f"{i:4d}/{r:8.2f}")
```

executed in 3ms, finished 14:22:04 2021-11-08

```
   5/    3.14
```

This fragment has the effect that the values of `i` and `r` are written to the screen as follows:

```
   5/    3.14
```

The value of `i` is written in `d` format, as int value, and the value of `r` is written in `f` format, as floating value. The symbols `d` and `f` originate respectively from "decimal", and "floating point" numbers. The numbers 4 respectively 8.2 denote that the integer value is written 4 positions wide (that is, 3 spaces and a " `5` " character), and that the floating value is written 8 positions wide, with 2 characters after the decimal point (that is, 4 spaces and the text " `3.14` "). A list of format specifiers is given in the table below:

Format specifier	Result
`:b`	Binary format
`:d`	Decimal format
`:10d`	Decimal format (at least 10 positions wide)
`:e`	Scientific format

Format specifier	Result
:f	Fix point number format
:9f	Fix point number format (at least 9 positions wide)
:.4f	Fix point number format (with 4 characters after the decimal point)
:9.4f	Fix point number format (at least 9 positions wide, and with 4 characters after the decimal point)
:g	General format
:n	Number format
:%	Percentage format

Finally, there are also a few special character sequences called *escape sequence* which allow to print characters like horizontal tab (which means jump to next tab position in the output"), or newline (which means "go to the next line in the output") in a format string. An escape sequence consists of two characters. First a backslash character `\`, followed by a second character. The escape sequence are presented below:

Escape sequence	Meaning
\n	New line
\t	Horizontal tab
\"	The character "
\\	The character \

An example is shown below. The value of `j` is written at the tab position, the output goes to the next line again at the first tab position, and outputs the value of `r`.

In [5]:

```
i = 5
j = 10
r = 3.14
print(f"{i:6d}\t{j:d}\n\t{r:.2f}\n")
```

executed in 3ms, finished 14:22:04 2021-11-08

```
     5  10
        3.14

```

5.2 Read and write files

Within Python, it is possible to read and write files. To do this, firstly the *open* function is used. An example to write to a file is shown below:

In [6]:

```
f = open("example.txt","w")

f.write("This is an example on how create and write data into a .txt file \nA new l
f.write("We can continue writing by calling the write function\n")

f.close()
```

executed in 113ms, finished 14:22:04 2021-11-08

```
---------------------------------------------------------------------------
----
OSError                                   Traceback (most recent call l
ast)
<ipython-input-6-37f9caec7ea5> in <module>
----> 1 f = open("example.txt","w")
      2
      3 f.write("This is an example on how create and write data into a
.txt file \nA new line can be created as well\n")
      4 f.write("We can continue writing by calling the write function
\n")
      5

OSError: [Errno 30] Read-only file system: 'example.txt'
```

In the above example, the `open` function is used to open/create a file called `"example.txt"`. The character `w` is used to indicate that the file should be overwritten, or newly created. The function `.write` is used to write new data into the opened file, defined by a string. Finally, the file can be closed with `.close()`.

Instead of overwriting a file, new data can also be appended to a file. In order to use the append mode, the character `a` should be used. Now, new data can be appended to the `example.txt` file, again using the `.write` function.

In [ ]:

```
f = open("example.txt","a")

f.write("This line will be appended to the file")

f.close()
```

executed in 214ms, finished 14:22:04 2021-11-08

Finally, a file can also be read, using the `.read` function:

In [ ]:

```
f = open("example.txt","r")

if f.mode == "r":
    contents = f.read()
    print(contents)

f.close()
```

executed in 200ms, finished 14:22:04 2021-11-08

In order to read the file, the character `r` should be used in the `open` function. Now, the function `.read()` can be used to read the contents of the file. For this, the file should be opened in read-mode, which is checked by `if f.mode=="r"` .

A more elaborate explanation on reading and writing files can be found here (https://www.guru99.com/reading-and-writing-files-in-python.html), however, this will not be used in the scope of this course.

5.3 Plotting

A nice way to get an overview of your data is to plot this. In the next paragraphs, different kind of plots in Python will be shortly elaborated, a more elaborate guide on how to plot can be found here (https://realpython.com/python-matplotlib-guide/).

The `matplotlib.pyplot` library is used to get these plots. This library can be imported by entering `import matplotlib.pyplot as plt` . All plotting options from the `matplotlib.pyplot` library can be found here (https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html). In this section, we will go into more depth on the line, step, bar, and histogram plot types (which are only a few of the plot types in the `matplotlib.pyplot` library), and how the style of the plot can be changed (titles, axis labels, etc.). A large gallery containing pretty much every example you would ever need can be found here (https://matplotlib.org/stable/gallery/index.html).

Below we show the most simple method of creating a simple line plot. We first create some data (using the `numpy` library), and then we plot it by first creating a figure using `plt.figure()` , then we draw a plot with two lines using `plt.plot(x, y1, x, y2)` .

In [ ]:

```
x = np.arange(0, 5, 0.2)
y1 = np.sin(x)
y2 = np.cos(x)

fig = plt.figure()
line = plt.plot(x, y1, x, y2)
```

executed in 167ms, finished 14:22:04 2021-11-08

Type *Markdown* and LaTeX: α^2

5.3.1 Line plot

The function `plt.plot` can be used to create a simple *line plot*. An example:

In [ ]:

```
x = np.arange(0, 5, 0.2)
y1 = np.sin(x)
y2 = np.cos(x)

fig, ax1 = plt.subplots()
plt.plot(x, y1, label='sin(x)')
plt.plot(x, y2, color='green', marker='o', linestyle='dashed',
                linewidth=2, markersize=5, label='cos(x)')
plt.xlabel("X-axis, e.g. Time [s]")
plt.ylabel("Y-axis, e.g. Distance [m]")
plt.title("Title for your plot")
plt.grid()
plt.legend(title='Parameter where:');
```

executed in 128ms, finished 14:22:04 2021-11-08

`plt.plot(x,y)` will just create a simple line. As shown above, the line can be adjusted by adding several inputs to `plt.plot` , for example:

- `color` , which changes the color of the line
- `marker` , which determines the shape of the markers used at every datapoint
- `linestyle` , which determines the linestyle, in this case (e.g. dashed)
- `linewidth` , which determines the linewidth
- `markersize` , which changes the size of the marker at every datapoint
- `label` , which is used to generate a legend for the plot

To create a new figure as output of a section, `fig, ax1 = plt.subplots()` can be used.

Furthermore, labels can be added to the axes of the graph, as well as a title and a legend. This can be done by the functions `plt.xlabel()` for the label on the x-axis, `plt.ylabel()` for the label on the y-axis, and `plt.title()` for the title of the plot. If labels are defined for lines in the plot (which is done by the input `label=`), `plt.legend()` can be used to add a legend to the plot.

Finally, to add a grid to the plot, `plt.grid()` can be used.

More information on the line plot can be found at
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html
(https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html)

5.3.2 Step plot

A *step plot* is similar to a line plot, only now the `plt.step()` function is used.

More information on the step plot can be found at
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.step.html#matplotlib.pyplot.step
(https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.step.html#matplotlib.pyplot.step)

In [ ]:

```
x = np.arange(0, 5, 0.2)
y1 = np.sin(x)
y2 = np.cos(x)

fig, ax1 = plt.subplots()
plt.step(x, y1, label='sin(x)')
plt.step(x, y2, color='green', marker='o', linestyle='dashed',
                linewidth=2, markersize=5, label='cos(x)')
plt.xlabel("X-axis, e.g. Time [s]")
plt.ylabel("Y-axis, e.g. Distance [m]")
plt.title("Title for your plot")
plt.grid()
plt.legend(title='Parameter where:');
```

executed in 87ms, finished 14:22:04 2021-11-08

5.3.3 Bar plot

A bar plot is used to present categorical data, with bars of which the height represents the frequency of each category. To create a *bar plot*, the function `plt.bar` can be used. For the bar plot, there are different inputs compared to the line plot. An example:

In [ ]:

```
x = ["Product 1","Product 2","Product 3","Product 4","Product 5"]
y = [5, 3, 5, 2, 9]

fig, ax1 = plt.subplots()
plt.bar(x, y, width=0.6, color='red')
plt.xlabel("X-axis, e.g. Product")
plt.ylabel("Y-axis, e.g. Occurance [-]")
plt.title("Title for your plot")
plt.grid();
```

executed in 68ms, finished 14:22:04 2021-11-08

For each bar, the width (w.r.t the x-axis) can be defined by `width=` . Furthermore labeling the x- and y-axes are the same as for line plots, as well as other layout-options for the graph.

More information on the bar plot can be found at
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.bar.html#matplotlib.pyplot.bar
(https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.bar.html#matplotlib.pyplot.bar)

5.3.4 Histogram plot

With a histogram, one can visualize the distribution of a dataset. To create a histogram, the function `plt.hist` is used. An example:

In [ ]:

```
mu, sigma = 100, 15
x = mu + sigma * random.randn(10000)

fig, ax1 = plt.subplots()
plt.hist(x, bins=50, density=True, color='blue')
plt.xlabel("X-axis, e.g. Time [hours]")
plt.ylabel("Y-axis, e.g. Probability [-]")
plt.title("Title for your plot")
plt.grid();
```

executed in 41ms, finished 14:22:04 2021-11-08

For this example, `x` contains data in which the numbers 1 to 6 occur in different amounts. This is visualized in the histogram using `plt.hist`, where the number of bins can be defined by `bins=` (if not defined, this will be 10 bins). The number of bins represents the number of bars that will be present in the plot. When the input `density=True` is used, the probability of each x-axis unit will be visualized, otherwise the occurence will be shown.

More information on the histogram plot can be found at
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.hist.html
(https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.hist.html)

To use the examples in this chapter, first run the code below to import the right libraries.

The distributions used in this chapter are retrieved from the `random` module from the `numpy` library.

In [1]:

```
# ================================
# Imports
# ================================
from numpy import random
from dataclasses import dataclass
from matplotlib import pyplot as plt
```

executed in 218ms, finished 14:24:20 2021-11-08

6 Modeling stochastic behavior

Many processes in the world vary a little bit each time they are performed. Setup of machines goes a bit faster or slower, patients taking their medicine takes longer this morning, more products are delivered today, or the quality of the manufactured product degrades due to a tired operator. Modeling such variations is often done with stochastic distributions. A distribution has a mean value and a known shape of variation. By matching the means and the variation shape with data from the system being modeled, an accurate model of the system can be obtained. We use the `numpy.random` library, which has many stochastic distributions available. This chapter explains how to use them to model a system, and lists a few commonly used distributions.

The following fragment illustrates the use of the random distribution to model a dice. Each value of the six-sided dice is equally likely to appear. Every value having the same probability of appearing is a property of the integer uniform distribution, in this case using interval $[1, 7)$ (inclusive on the left side, exclusive on the right side).

In the fragment, the distribution `dice` is defined using a lambda function of the `numpy.random` random integer function `randint` (more on the `randint` later in section 6.2). We simulate two dice throws by taking two samples of this distribution by calling `dice()` .

In [2]:

```
dice = lambda: random.randint(1,7)  # Defining the distribution through a lambda fu

# Taking two random samples
dice_throw1 = dice()
dice_throw2 = dice()

print(f"The dice land on {dice_throw1} and {dice_throw2}.")
```

executed in 4ms, finished 14:24:20 2021-11-08

```
The dice land on 4 and 2.
```

6.1 Lambda functions

Lambda functions are like a simplified version of normal python functions. While other functions are defined using the `def` keyword, the lambda function is defined using the `lambda` keyword. There are many reasons for using lambda functions. However, in this course, we will use lambda functions for one purpose only: to define stochastic distributions. If you want to find out about other uses of lambda function click here (https://realpython.com/python-lambda/).

Below we find the most simple example of defining a stochastic distribution with the lambda function. A constant distribution `u` is defined, which when sampled returns `3`. We can take a sample by calling `u()`. More will be explained on constant distributions and other distributions in the next section.

In [3]:

```
u = lambda: 3  # Defining the constant distribution
y = u()        # Taking a sample
print(f"The value {y} was sampled.")
```

executed in 4ms, finished 14:24:20 2021-11-08

```
The value 3 was sampled.
```

Lambda functions are defined as `lambda <input parameters> : <output function>`. As you can see, they can also accept input parameters.

Below is an example in which we define a function for throwing an n-sided dice, which takes `n` as input, and returns a random integer between `1` up to and including `n`.

In [4]:

```
dice = lambda n: random.randint(1,n+1)     # We define a function for throwing an x
dice_throw = dice(20)                      # We sample throwing a 20-sided dice
print(f"The dice lands on {dice_throw}.")
```

executed in 5ms, finished 14:24:20 2021-11-08

```
The dice lands on 9.
```

6.2 Distributions

In our models we use both constant, discrete and continuous distributions. A discrete distribution is a distribution where only specific values can be drawn, for example throwing a dice gives an integer number. A continuous distribution is a distribution where a value from a continuous range can be drawn, for example assembling a product takes a positive amount of time. The constant distributions are discrete distributions that always return the same value. They are useful during the development of the model. A list of distributions and information on how to use them can be found here (https://numpy.org/doc/1.16/reference/routines.random.html).

6.2.1 Constant distributions

When developing a model with stochastic behavior, it is hard to verify whether the model behaves correctly, since the stochastic results make it difficult to predict the outcome of experiments. As a result, errors in the model may not be noticed, they hide in the noise of the stochastic results. One solution is to first write a model without stochastic behavior, verify that model, and then extend the model with stochastic sampling. Extending the model with stochastic behavior is however an invasive change that may introduce new errors. These errors are again hard to find due to the difficulties to predict the outcome of an experiment. The constant distributions aim to narrow the gap by reducing the amount of changes that need to be done after verification.

With constant distributions, a stochastic model with sampling of distributions is developed, but the stochastic behavior is eliminated by temporarily using constant distributions. The model performs stochastic sampling of values, but with predictable outcome, and thus with predictable experimental results, making verification easier. After verifying the model, the constant distributions are replaced with the distributions that fit the mean value and variation pattern of the modeled system, giving a model with stochastic behavior. Changing the used distributions is however much less invasive, making it less likely to introduce new errors at this stage in the development of the model.

An example of a constant distribution was already shown before, but here it is repeated:

In [5]:

```
u = lambda: 3  # Defining the constant distribution
y = u()        # Taking a sample
print(f"The value {y} was sampled.")
```

executed in 6ms, finished 14:24:20 2021-11-08

```
The value 3 was sampled.
```

6.2.2 Discrete distributions

Discrete distributions return values from a finite fixed set of possible values as answer. In the `numpy.random` library, there are a few discrete distributions we can sample from: `randint` (uniform discrete), `binomial` (which can also be used for the bernoulli distribution), and `choice` (choosing randomly from a list). Below we show examples of these distributions.

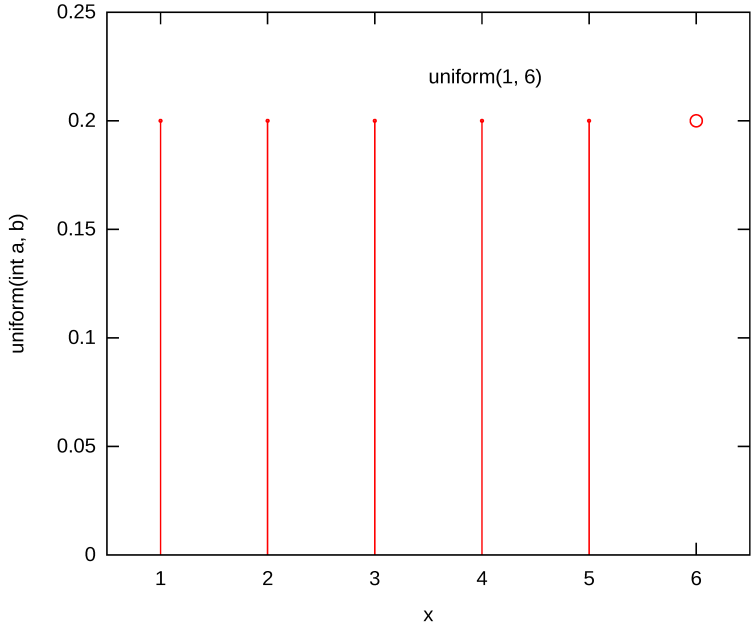
Discrete uniform

The `random.randint(a, b)` function allows us to sample from a uniform discrete distribution that has several equally likely outcomes for the numbers `{a, a+1, a+2, ..., b-2, b-1}`. Note that `b` is not included. More information and options can be found here (https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html).

- Function: `random.randint(a, b)`
- Parameters:
  - `a` : lower bound
  - `b` : upper bound (exclusive!)
- Mean: $\frac{a+b-1}{2}$

- Variance: $\frac{(b-a)^2-1}{12}$

Figure 6.2: Discrete uniform distribution



An example of `randint` was shown in the 6-sided dice example, which is repeated below. It has a lower bound of `1`, and an upper bound of `7`.

In [6]:

```
dice = lambda: random.randint(1,7)  # Defining the distribution through a lambda fu

# Taking two random samples
dice_throw1 = dice()
dice_throw2 = dice()

print(f"The dice land on {dice_throw1} and {dice_throw2}.")
```

executed in 5ms, finished 14:24:20 2021-11-08

```
The dice land on 1 and 4.
```

Binomial distribution

The binomial distribution `binomial(n, p)` allows us to sample from a discrete distribution which is used to model the outcome of an experiment with `n` trials, each with `p` chance of succes. More information and options can be found here (https://numpy.org/doc/stable/reference/random/generated/numpy.random.binomial.html).

- Function: `random.binomial(n,p)`
- Parameters:
  - n: number of trials

    - p: chance of succes for each trial
- Mean: np
- Variance: $np(1-p)$

Below we show an example of the binomial distribution, in which we model the flipping of a coin `10` times, with `p = 0.5` probability of succes (heads).

In [7]:

```
p = 0.5  # probability of heads
coin_flipping = lambda n : random.binomial(n, p)  # Defining the binomial distribut

N = 10
heads = coin_flipping(N)
print(f"We flip a coin {N} times, it lands on head {heads} times.")
```

executed in 5ms, finished 14:24:20 2021-11-08

```
We flip a coin 10 times, it lands on head 7 times.
```

Bernoulli distribution

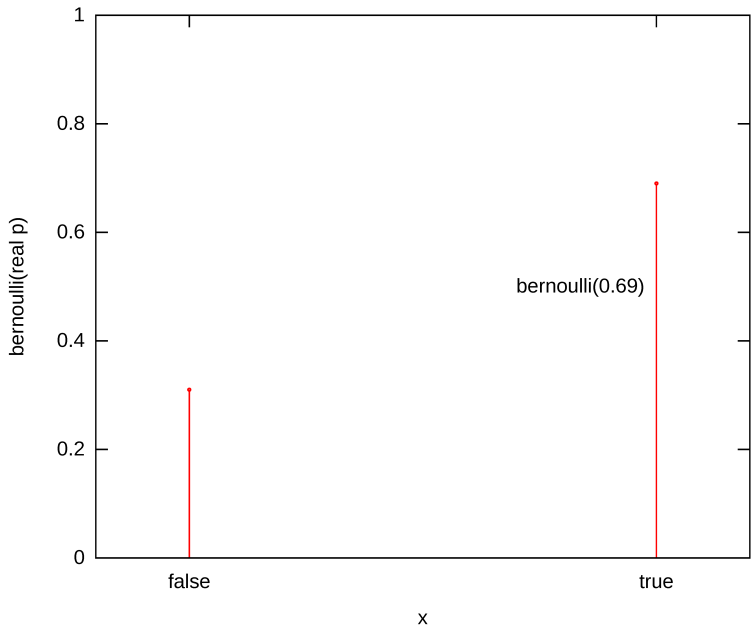
A special case of the Binomial distribution is the Bernoulli distribution, which is the case in which `n = 1`, and thus has two possible outcomes: 0 and 1. The bernoulli distribution can be samples from with `random.binomial(1, p)`.

- Function: `random.binomial(1,p)`
- Parameters:
    - p: chance of succes for each trial
- Mean: p
- Variance: $p(1-p)$

This is a useful distribution for sampling behaviour of which the outcomes is either true or false.

Figure 6.1: Bernoulli distribution

Figure 6.1: Bernoulli distribution



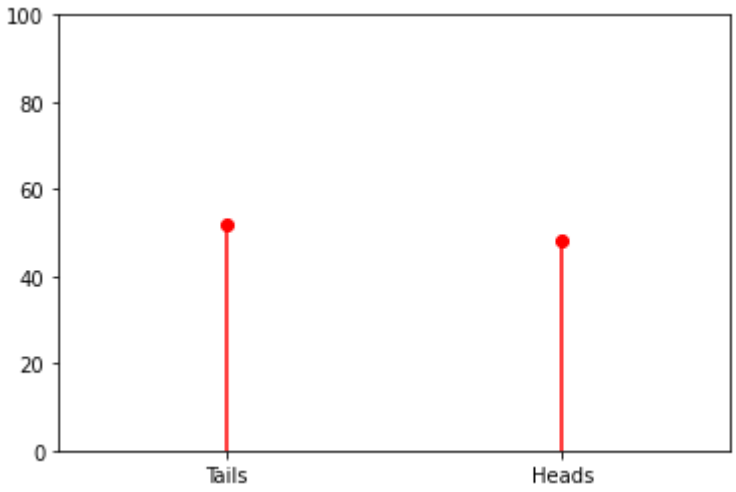
Below is an example in which we plot the outcome of taking 100 samples of a coin flip.

In [8]:

```
p = 0.5
N_samples = 100
samples = random.binomial(1, p, N_samples)  # We take N samples of the bernoulli di
n_heads = sum(samples)
n_tails = N_samples - sum(samples)
print(f"The result of the coin flips is: {n_heads} times heads, {n_tails} times tai

# Creating a stem plot
fig, ax1 = plt.subplots();
ax1.stem(["Tails", "Heads"], [n_tails, n_heads], basefmt=" ", linefmt = "r-", marke
ax1.set_ylim(0, N_samples);
ax1.set_xlim(-0.5, 1.5);
```

executed in 151ms, finished 14:24:20 2021-11-08

```
The result of the coin flips is: 48 times heads, 52 times tails.
```



Random choice

The `random.choice(l)` function allows us to rake a random sample from a list `l`. The function assumes assumes a uniform distribution; each entry in the list has the same probability. More information and options can be found here (https://numpy.org/doc/stable/reference/random/generated/numpy.random.choice.html).

- Function: `random.choice(l)`
- Parameters:
  - l: list from which a sample is taken

In [9]:

```
l = [2, 3.0, "a string"]                  # The list from which we choose randomly
RandomChoice = lambda : random.choice(l)  # Defining the binomial distribution for

sample = RandomChoice()
print(f"We sample {sample}.")
```

executed in 4ms, finished 14:24:20 2021-11-08

```
We sample 2.
```

6.2.3 Continuous distributions

Continuous distributions return a value from a continuous range.

Continuous uniform

The `random.uniform(a, b)` function allows us to sample from a continuous distribution with equal chance of sampling each value in the range $[a,b)$. Note that b is not included. More information and options can be found here (https://numpy.org/doc/stable/reference/random/generated/numpy.random.uniform.html).

- Function: `random.uniform(a, b)`
- Parameters:
  - `a` : lower bound
  - `b` : upper bound (exclusive!)
- Mean: $\frac{a+b}{2}$
- Variance: $\frac{(b-a)^2}{12}$

Figure 6.3: Continuous uniform distribution

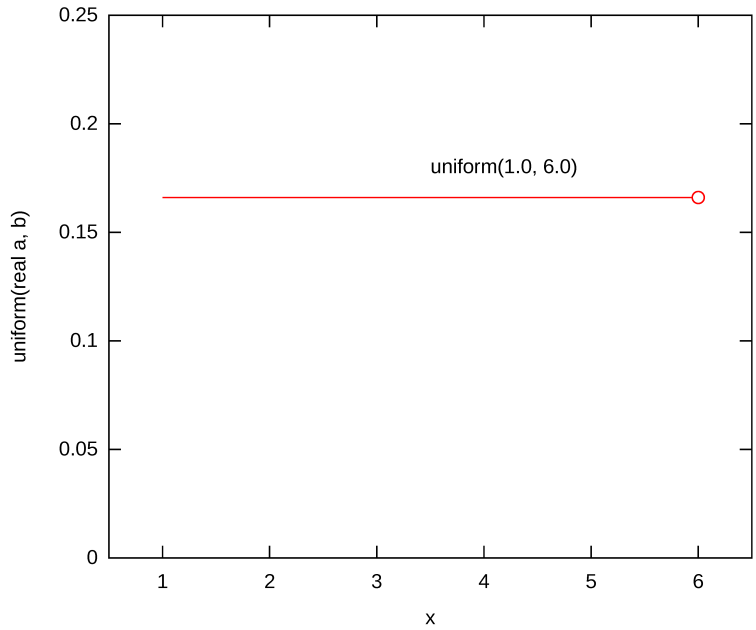


Figure 6.3: Continuous uniform distribution

Below is an example:

In [10]:

```
a = 1 # lower bound
b = 5 # upper bound
distribution = lambda : random.uniform(a,b)

sample = distribution()
print(f"We sample {sample:.4f}.")
```

executed in 5ms, finished 14:24:20 2021-11-08

```
We sample 3.3575.
```

Gamma

The `random.gamma(a, b)` function allows us to sample from a gamma distribution which has either a decreasing probability function, or a peak. More information and options can be found here (https://numpy.org/doc/stable/reference/random/generated/numpy.random.gamma.html).

- Function: `random.gamma(a, b)`
- Parameters:
  - a : shape parameter
  - b : scale parameter
- Mean: ab
- Variance: ab^2

Figure 6.4: Gamma distribution

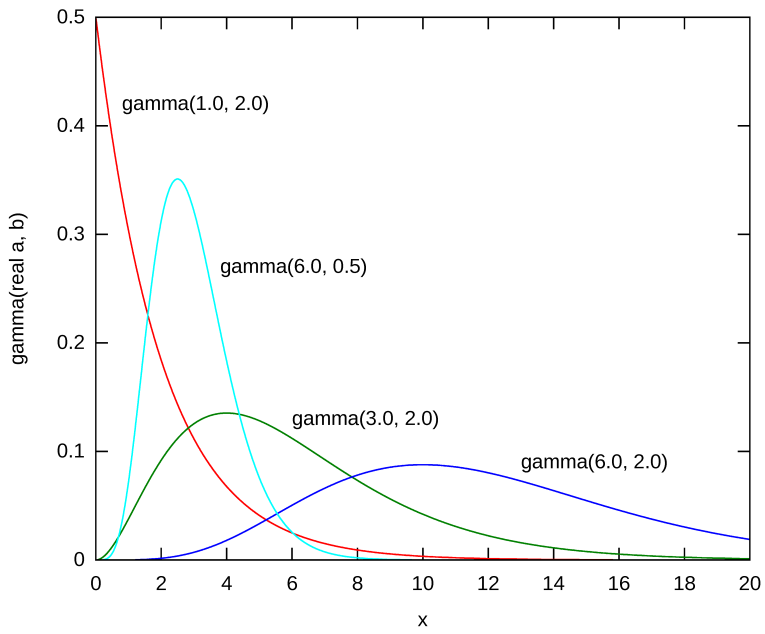


Figure 6.4: Gamma distribution

Below is an example:

In [11]:

```
a = 6.0 # shape parameter
b = 0.5 # scale parameter
u = lambda : random.gamma(a,b)

sample = u()
print(f"We sample {sample:.4f}.")
```

executed in 4ms, finished 14:24:20 2021-11-08

```
We sample 1.5595.
```

Normal distribution

The `random.randn()` function generates a random float sampled from a univariate “normal” (Gaussian) distribution of mean 0 and variance 1. More information and options can be found here (https://numpy.org/doc/stable/reference/random/generated/numpy.random.gamma.html). For random samples from $N(\mu,\sigma^2)$ use the following function:

- Function: `sigma * random.randn() + mu`
- Parameters:
  - `mu` : mean
  - `sigma` : standard deviation
- Mean: μ (`mu`)
- Variance: σ^2 (`sigma`)

Below is an example:

In [12]:

```
mu = 10 # mean
sigma = 2 # standard deviation
distribution = lambda : sigma * random.randn() + mu

sample = distribution()
print(f"We sample {sample:.4f}.")
```

executed in 4ms, finished 14:24:20 2021-11-08

```
We sample 11.1341.
```

Exponential distribution

The `random.exponential(a)` function allows us to draw samples from an exponential distribution. Its probability density function is $ a e^{-ax}$, for x > 0 and 0 elsewhere. a is the scale parameter. More information and options can be found here (https://numpy.org/doc/stable/reference/random/generated/numpy.random.exponential.html).

- Function: `random.exponential(a)`
- Parameters:
  - `a` : scale parameter
- Mean: $\frac{1}{a}$
- Variance: $\frac{1}{a^2}$

Below is an example:

In [13]:

```
a = 2
distribution = lambda : random.exponential(a)

sample = distribution()
print(f"We sample {sample:.4f}.")
```

executed in 3ms, finished 14:24:20 2021-11-08

```
We sample 1.6859.
```

Poisson distribution

The function `random.poisson(lambda)` allows us to draw samples from a Poisson distribution. The Poisson distribution expresses the probability of a given number of events occurring in a fixed interval of time or space if these events occur with a known constant mean rate and independently of the time since the last event. More information and options can be found here (https://numpy.org/doc/stable/reference/random/generated/numpy.random.poisson.html).

- Function: `random.poisson(lambda)`
- Parameters:
  - `lambda` : expected number of events occurring in a fixed-time interval
- Mean: λ (`lambda`)
- Variance: λ (`lambda`)

Below is an example:

In [14]:

```
lambda = 3
distribution = lambda : random.poisson(lambda)

sample = distribution()
print(f"We sample {sample:.4f}.")
```

executed in 5ms, finished 14:24:20 2021-11-08

```
  File "<ipython-input-14-b46162483a88>", line 1
    lambda = 3
           ^
SyntaxError: invalid syntax
```

6.3 Simulating stochastic behavior

In this chapter, the mathematical notion of stochastic distribution is used to describe how to model stochastic behavior. Simulating a model with stochastic behavior at a computer is however not stochastic at all. Computer systems are deterministic machines, and have no notion of varying results.

A (pseudo-)random number generator is used to create stochastic results instead. It starts with an initial *seed*, an integer number (you can give one at the start of the simulation). From this seed, a function creates a stream of `random` values. When looking at the values there does not seem to be any pattern. It is not truly random however. Using the same seed again gives exactly the same stream of numbers. This is the reason to call the function a `pseudo` -random number generator (a true random number generator would never produce the exact same stream of numbers). A sample of a distribution uses one or more numbers from the stream to compute its value. The value of the initial seed thus decides the value of all samples drawn in the simulation.

While doing a simulation study, performing several experiments with the same initial seed invalidates the results, as it is equivalent to copying the outcome of a single experiment a number of times. On the other hand, when looking for the cause of a bug in the model, performing the exact same experiment is useful as outcomes of previous experiments should match exactly.

6.3.1 Modelling the coincidence problem

Let us create our first model.

There are two strangers sitting next to each other in the bus. They start talking to each other. To their surprise, they find out they have a friend in common. What a coincidence! We'd like to calculate the odds of this occuring.

Our model has as input:

- `frA` : the number of friends of person A
- `frB` : the number of friends of person B
- `population` : the population of the city they both live in.

The model to calculate if they have a friend in common is shown below. It returns True if the strangers have a friend in common, and false if they do not.

In [15]:

```
def model(frA, frB, population):
    u = lambda: random.randint(1, population)
    s = []
    for j in range(0, frB):  # generate friends of B
        while True:
            k = u()
            if k not in s:
                break
        s = s + [k];
        if k <= frA:
            return True
    return False
```

executed in 5ms, finished 14:24:20 2021-11-08

Suppose that both person A and B have 200 friends, are living in Eindhoven which has a population of 220000. Lets run the model and see if they have a friend in common.

In [16]:

```
# Running the model
frA = 200
frB = 200
population = 220000
common = model(frA, frB, population)
print(f'There {"is" if common else "is not"} a common acquaintance.')
```

executed in 6ms, finished 14:24:20 2021-11-08

```
There is not a common acquaintance.
```

Now obviously, one simulation does not tell much about the odds of two strangers having a common friend. Lets make it an experiment, in which we run the model multiple times:

In [17]:

```
def experiment(N):
    population = 220000
    frA = 200
    frB = 200

    success = 0
    for j in range(0, N):
        common = model(frA, frB, population)
        if common:
            success = success + 1
    return success
```

executed in 5ms, finished 14:24:20 2021-11-08

Processing math: 100%

If we run the model 10000 times:

In [*]:

```
# Running the experiment
N = 10000
success = experiment(N)
print('Probability of common acquaintance = %g' % (success/N))
```

execution queued 14:24:20 2021-11-08

6.4 Exercises

Exercise 6.4.1

Accoriding to the manual, for a gamma distribution with parameters `(a,b)`, the mean equals ab.

(a) Verify if this is true for three different pairs of `a` and `b`.

(b) How many samples from the distribution are approximately required to determine the mean up to three decimals accurate?

In [ ]:

```
# Exercise 6.4.1
```

Exercise 6.4.2

Estimate the mean μ and variance σ^2 of a triangular distribution triangle$(1, 2, 5)$ by simulating 1000 samples. (Recall that the variance σ^2 of n samples x_i can be calculated by: $\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$.)

In [ ]:

```
# Exercise 6.4.2
```

Exercise 6.4.3

We would like to build a small game, called Higher or Lower. The computer picks a random integer number between 1 and 14. The player then has to predict whether the next number will be higher or lower. The computer picks the next random number and compares the new number with the previous one. If the player guesses right his score is doubled. If the player guesses wrong, he looses all and the game is over. Try the following specification.

In [ ]:

```
def HoL():
    u = lambda: random.randint(1, 15)
    sc = 1
    c = True
    oldval = 0

    new = u()
    print("Your score is: ", sc)
    print("The computer drew: ", new)

    while c:
        s = input('(h)igher or (l)ower: ')
        oldval = new
        new = u()
        print("The computer drew: ", new)
        if new == oldval:
            c = False
        else:
            c = (new > oldval) == (s == "h")
        if c:
            sc = 2*sc
        else:
            sc = 0
        print("Your score is: ", sc)

    print("GAME OVER...")
```

In [ ]:

```
HoL()
```

(a) What is the begin score?

(b) What is the maximum end score?

(c) What happens, when the drawn sample is equal to the previous drawn sample?

(d) Extend this game specification with the possibility to stop.

In [ ]:

```
# exercise 3d
def HoL():
    ...
```

In [ ]:

```
HoL()
```

6.5 Answers to exercises

Answer to 6.4.1

[Click for the answer to 6.4.1]

Answer to 6.4.2

[Click for the answer to 6.4.2]

Answer to 6.4.3

[Click for the answer to 6.4.3]

To use the examples in this chapter, first run the code below to import the right libraries.

This is the first time we introduce the PyCh simulation library.

In [1]:

```
# ================================
# Imports
# ================================
from PyCh import *
from numpy import random
from dataclasses import dataclass
import math
```

executed in 356ms, finished 14:24:55 2021-11-08

```
PyCh version 1.0 imported succesfully.
```

7 Processes

The PyCh simulation library has been designed for modeling and analyzing systems with many components, all working together to obtain the total system behavior. Each component exhibits behavior over time. Sometimes they are busy making internal decisions, sometimes they interact with other components. The language uses a process to model the behavior of a component (the primary interest are the actions of the component rather than its physical representation). This leads to models with many processes working in parallel (also known as concurrent processes), interacting with each other.

Another characteristic of these systems is that the parallelism happens at different scales at the same time, and each scale can be considered to be a collection of co-operating parallel working processes. For example, a factory can be seen as a single component, it accepts supplies and delivers products. However, within a factory, you can have several parallel operating production lines, and a line consists of several parallel operating machines. A machine again consists of parallel operating parts. In the other direction, a factory is a small element in a supply chain. Each supply chain is an element in a (distribution) network. Depending on the area that needs to be analyzed, and the level of detail, some scales are precisely modeled, while others either fall outside the scope of the system or are modeled in an abstract way.

In all these systems, the interaction between processes is not random, they understand each other and exchange information. In other words, they communicate with each other. PyCh uses channels to model the communication. A channel connects a sending process to a receiving process, allowing the sender to pass messages to the receiver. This chapter discusses parallel operating processes only, communication between processes using channels is discussed in Chapter 8.

As discussed above, a process can be seen as a single component with behavior over time, or as a wrapper around many processes that work at a smaller scale. PyCh supports both kinds of processes.

7.1 The simulation environment

When we create a model in PyCh, we must first define the simulation environment in which the processes of this model live. We typically do so by starting our model with: `env = Environment()`, which creates instance `env` of the `Environment` class. After creating an environment, the next step is to define the processes living in this environment (more on this in the next section). When all processes have been defined, we can simulate an instance of the `Environment` class by calling its `run()` function, so for environment `env` this is done through `env.run()`. When we run an environment, the processes in the environment start running. The simulation stops when all processes have finished executing.

Below is an example of how we generally start when creating a model. We can execute this model, but nothing will happen, as our environment does not yet contain any processes.

In [2]:

```
def model():  # we define our model
    env = Environment()  # we define its simulation environment

    '''
    The processes in this simulation environment go here
    '''

    env.run() # we run the simulation environment

model() # we execute our model
```

executed in 4ms, finished 14:24:55 2021-11-08

7.1.1 Simulation time

When we talk about the simulation environment, we must also explain the concept of *simulation time*. With simulation time, we mean the time inside the simulation environment. This is not the same as real-time! A simulation of a lengthy process spanning days, months or even years in simulation time, could only take a few seconds to execute in real-time! In our models, the current simulation time can be accessed through the `Environment.now` variable.

When we run a simulation, the default option is to run until all processes have finished executing. However, we can also run our environment until simulation time `t` using `Environment.run(until=t)`, or we can run it until a process `P` has finished executing using `Environment.run(until=P)`. How we define such a process is explained in the next section.

The example below shows for environment `env` how `env.run(until=t)` and `env.now` can be utilized. Again, there are no processes, so the simulation ends without anything happening after 5 units of simulation time. Note: these units of simulation time are whatever you define them to be. One unit can represent 1 second, but it can also represent 15 minutes, or 7 days, or whatever you want it to.

In [3]:

```
def model():
    env = Environment()

    '''
    The processes in this simulation environment go here
    '''

    env.run(until=5) # we run the simulation environment until time = 5
    print(f"The simulation ends after {env.now} days.")  # we print the simulation

model()
```

executed in 5ms, finished 14:24:55 2021-11-08

```
The simulation ends after 5 days.
```

7.2 A single process

After defining the simulation environment, the next step is to define the processes living in this environment. In Pych processes are defined using process functions, which are denoted by the decorator `@process` above the function definition. A process function must always pass as the simulation environment `env` in which the process lives as one of its arguments.

A Pych process can generate events and can `yield` these events. The `yield` statement indicates that the process is suspended till the event has been triggered, after which the process continues. A process must always yield at least one event, so in the next subsection we will introduce the first type of event that will be used.

7.2.1 The timeout event

The first event type that we introduce is the `Environment.timeout(t)` event. This event denotes a timeout of `t` in the environment's simulation time.

Below we extend our model with its first process. We define process function `P` with the simulation environment `env` as its only argument. Process `P` contains three statements. The first is `delay = env.timeout(2.5)`, which indicates that we create a `delay` event that will trigger after `2.5` units in simulation time. The second is `yield delay`, which indicates that the process waits till the `delay` event has triggered, after which it continues. The third statement is the `print` statement, in which we use the `env.now` function to show what the current simulation time is.

The model below contains one instance (`P1`) of the process function (`P`).

In [4]:

```
@process
def P(env):
    delay = env.timeout(2.5)  # we define the timeout event
    yield delay  # The process waits till the timeout event is triggered
    print(f"Hello. I am a process which is finished after {env.now} hours.")

def model():
    env = Environment()
    P1 = P(env)  # Our model has one instance of process P
    env.run()

model()
```

executed in 8ms, finished 14:24:55 2021-11-08

```
Hello. I am a process which is finished after 2.5 hours.
```

A model environment can contain multiple instances created using the same process definition. When simulating the environment using `env.run()`, the simulation continues until both processes are finished executing. To demonstrate, below is an example of a model with two processes `P1` and `P2`.

In the same example, we also show with `yield env.timeout(1)` that we can create an event, and yield it on the same line.

In [5]:

```
@process
def P(env, i):
    yield env.timeout(1)  # we create an event and yield it on the same line
    print(f"I am process {i}")

def model():
    env = Environment()
    P1 = P(env, 1)  # our first instance of process P
    P2 = P(env, 2)  # our second instance of process P
    env.run()

model()
```

executed in 49ms, finished 14:24:55 2021-11-08

```
I am process 1
I am process 2
```

Note that when we define a timeout event, its timer starts ticking as soon as the event is defined. The `yield` statement indicates that the process waits till the event has been triggered. If the event had already started earlier, or has already been triggered, then the process does not wait the entire timeout duration. In the example below we can see how this works in practice.

As you can see in the example, the clock of the timeout event starts ticking when the event is defined. When Process `P` encounters the first yield statement it is suspended till the timeout event `delay1` is triggered at `t=0.5`, then it is suspended again till `delay2` is triggered at `t=1.0` (so not at `t=1.5`!). Finally, at the third yield statement, process `P` does not suspend, instead it continues right away, because event `delay3` had already been triggered in the past.

In [6]:

```
@process
def P(env):
    delay1 = env.timeout(0.5)  # this event is triggered after 0.5 seconds
    delay2 = env.timeout(1.0)  # this event is triggered after 1.0 seconds
    delay3 = env.timeout(0.5)  # this event is also triggered after 0.5 seconds
    yield delay1
    print(f"The first timeout event was triggered at time {env.now:.1f}")
    yield delay2
    print(f"The second timeout event was triggered at time {env.now:.1f}")
    yield delay3
    print(f"The third timeout event had already been triggered, so the process cont

def model():
    env = Environment()
    P1 = P(env)
    env.run()

model()
```

executed in 11ms, finished 14:24:55 2021-11-08

```
The first timeout event was triggered at time 0.5
The second timeout event was triggered at time 1.0
The third timeout event had already been triggered, so the process cont
inues right away at time 1.0
```

7.3 Process in process

A process can create other processes, and it can even wait until another process has finished through the `yield` statement (although it can also continue unhalted). In the example below main process `P` creates sub processes `P1` and `P2`, and waits till they have finished executing to continue. The concept of 'a process in a process' is very useful in keeping the model structured.

In [7]:

```
@process
def main_process(env):
    print("Start process 1")
    P1 = sub_process(env, 1)  # we start process P1
    yield P1  # we wait untill process P1 has finished executing
    print("Start process 2")
    P2 = sub_process(env, 2)  # we start process P2
    yield P2  # we wait untill process P2 has finished executing

@process
def sub_process(env, i):
    yield env.timeout(1)
    print(f"Finished process {i}")

def model():
    env = Environment()
    P = main_process(env)
    env.run()

model()
```

executed in 8ms, finished 14:24:55 2021-11-08

```
Start process 1
Finished process 1
Start process 2
Finished process 2
```

Just like with the timeout event, a process starts running when it is defined, not when we yield it. Suppose we redefine the main process as seen below. If we run our model again, we can see that the order of execution is changed.

In [8]:

```
@process
def main_process(env):
    print("Start process 1")
    P1 = sub_process(env, 1) # we start process P1
    print("Start process 2")
    P2 = sub_process(env, 2) # we start process P2
    yield P1 # we wait untill process P1 has finished executing
    yield P2 # we wait untill process P2 has finished executing

model()
```

executed in 4ms, finished 14:24:55 2021-11-08

```
Start process 1
Start process 2
Finished process 1
Finished process 2
```

7.4 Many processes

Some models consist of many similar processes. In Python, we can utilize list comprehension to quickly create many processes. Below is an example of using list comprehension to create 10 instances of `P` at once.

In [9]:

```
@process
def P(env, i):
    yield env.timeout(1)
    print(f"I am process {i}")

def model():
    env = Environment()
    Processes = [P(env, i) for i in range(10)]  # we use list comprehension to crea
    env.run()

model()
```

executed in 6ms, finished 14:24:55 2021-11-08

```
I am process 0
I am process 1
I am process 2
I am process 3
I am process 4
I am process 5
I am process 6
I am process 7
I am process 8
I am process 9
```

7.5 Process output

A process can `return` an output (e.g. `return main_output` and `return sub_output` in the example below). This can be utilized in two ways.

Firstly, when one process yields another process, it can use the output of that other process. For example, `sub_output = yield P1` in the example below. Alternatively, we can access the Process' property `Process.value` to access its output (e.g. `sub_output_alternative = P1.value` in the example below). Both methods result in the same output.

Secondly, we can use `Process.value` to retrieve output from our process after the model has finished simulating. For example, `main_output = P.value` in the example below. This can be very useful when we want to retrieve information from our processes after our model has finished executing (e.g. the throughput of a machine process).

In [10]:

```
@process
def main_process(env):
    P1 = sub_process(env)
    sub_output = yield P1
    sub_output_alternative = P1.value
    print(f"Our subprocess returns {sub_output}, or using the alternative method, a
    main_output = sub_output*2
    return main_output

@process
def sub_process(env):
    yield env.timeout(1)
    sub_output = 1
    return sub_output

def model():
    env = Environment()
    P = main_process(env)
    env.run()
    main_output = P.value
    print(f"The main process returns {main_output}")

model()
```

executed in 5ms, finished 14:24:55 2021-11-08

```
Our subprocess returns 1, or using the alternative method, also: 1.
The main process returns 2
```

7.5 Summary

- A process represents the behavior of a component in the simulation environment. An environment can have multiple processes running in parallel.
- The simulation environment is defined using `env = Environment()`.
  - The simulation environment can be run until all of its processes have finished executing using `env.run()`.
  - The simulation environment can be run until simulation time `t` using `env.run(until=t)`.
  - The simulation environment can be run until one of its processes has finished executing using `env.run(until=Process)`.
  - The current simulation time can be accessed using `env.now`.
- A process function specified with the `@process` decorator above it, and it is defined using `def P(env, ...)` with its simulation environment as its first argument.
  - A process can be instantiated by calling the process function, e.g. `P1 = P(env, ...)`.
  - Multiple processes can be instantiated from the same process function, e.g. `P1 = P(env, 1)` and `P2 = P(env, 2)`.
  - Many similar processes can be instantiated using list comprehension, e.g. `Processes = [P(env, i) for i in range(10)]`.
  - A process can instantiate other processes.
- A process can create and `yield` events through its lifetime.

    - One type of event is the timeout event `timeout_event = env.timeout(t)`, which is an event that is triggered after `t` simulation time.
    - Other events are communication events, which are explained in the next chapter.
    - A process can yield these events through `yield Event`, which means that the process will be suspended until the event has been triggered, after which the process continues.
    - A process can yield other processes through `yield Process`, which means that the process will be suspended until the other process has finished executing.
- A process can `return` an output when it has finished.
    - This output can be retrieved by other processes through `output = yield Process` or `Process.value`
    - This output can be retrieved when the model has finished executing through `Process.value`.

In [ ]:

To use the examples in this chapter, first run the code below to import the right libraries.

In [1]:

```
# ================================
# Imports
# ================================
from PyCh import *
from numpy import random
from dataclasses import dataclass
import math
```

executed in 395ms, finished 14:25:22 2021-11-08

```
PyCh version 1.0 imported succesfully.
```

8 Channels

In the previous chapter processes have been introduced. This chapter describes channels. A channel connects two processes and is used for the transfer of data or just signals. One process is the sending process, the other process is the receiving process. Communication between the processes takes place instantly when both processes are willing to communicate, this is called *synchronous* communication.

8.1 A channel

So now that you know what a channel is, you need to know how to use them in your models. You can add a channel to environment `env` by instantiating `Channel(env)`. A channel always has the simulation environment in which it operates as its single argument.

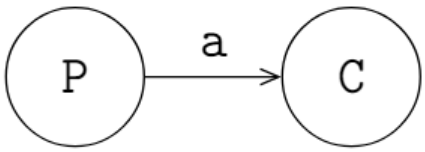
To communicate across a channel we use *communication events*. We can send an entity over a channel by using the communication event `Channel.send(entity)`, and we can receive over the same channel by using the communication event `Channel.receive()`. Unlike timeout events or processes, communication events do not start as soon as they are defined. Instead, `Environment.execute(communication event)` is used to start the communication event. A channel can transmit any object as entity; for example an integer, a string, or a custom data type.

Communication over a channel requires a process which is sending, and a process which is receiving. If a process has no other process to communicate with, then it will have to wait. Communication is only completed once the receiving process has received the entity from the sending process. If a process uses the `yield` statement, such as with `yield Environment.execute(communication event)`, then the process waits till communication has completed (from both sides!) before it continues.

8.1.1 Producer and consumer example

The figure below shows the two processes `P` and `C`, connected by channel `a`. Processes are denoted by circles, and channels are denoted by directed arrows in the figure. The arrow denotes the direction of communication. Process `P` is the sender or producer, process `C` is the receiver or consumer.

Figure 8.1: A producer and a consumer



In the model below, the producer `P` sends a stream of five integer values to consumer `C`. The producer sends integers `i` over channel `a`, which is realized through `env.execute(a.send(i))`. It sends the sequence of values `0, 1, 2, 3, 4` one after the other. The consumer receives these values one by one over channel `a` with `i = yield env.execute(a.receive())`. It then prints these values as output.

The simulation runs until there are no more events to execute. So in the example below the simulation will end once the producer has sent all five integers, even though the process of the consumer is supposed to go on forever (on account of its `while True:` loop). This is because none of the processes are active; all processes are either finished (the producer), or are waiting for other processes to do something (the consumer).

In [2]:

```
@process
def producer(env, c_out):
    for i in range(5):
        yield env.execute(c_out.send(i))

@process
def consumer(env, c_in):
    while True:
        i = yield env.execute(c_in.receive())
        print(i)

def model():
    env = Environment()
    a = Channel(env)
    P = producer(env, a)
    C = consumer(env, a)

    env.run()
    print("The simulation has finished.")

model()
```

executed in 49ms, finished 14:25:22 2021-11-08

```
0
1
2
3
4
The simulation has finished.
```

8.1.2 Synchronization signals

Earlier we said that we can transmit any type of entity over a channel. What we did not yet tell, is that it is also possible to send an empty signal by using `Channel.send(None)`, these signals are called *synchronization signals*, as they just synchronize actions between different processes without transmitting data.

Below we see an example in which synchronization signals are used; the producer does not send any data (it sends `None`). We first define our communication events through `sending = a.send(None)` and `receiving = a.receive()`, before executing and yielding them. This also allows us to later access the received entity through `receiving.entity` (which in this example is `None`).

In [3]:

```
@process
def producer(env, c_out):
    for i in range(5):
        sending = c_out.send(None)
        yield env.execute(sending)

@process
def consumer(env, c_in):
    while True:
        receiving = c_in.receive()
        yield env.execute(receiving)
        print(receiving.entity)

def model():
    env = Environment()
    a = Channel(env)
    P = producer(env, a)
    C = consumer(env, a)

    env.run()

model()
```

executed in 7ms, finished 14:25:22 2021-11-08

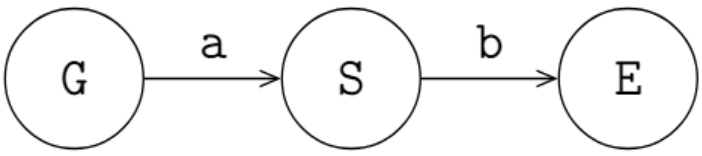
```
None
None
None
None
None
```

8.2 Two channels

A process can have more than one channel, allowing interaction with several other processes.

The next example shows two channels, `a` and `b`, and three processes, generator `G`, server `S` and exit `E`. Process `G` is connected via channel `a` to process `S` and process `S` is connected via channel `b` to process `E`. The model is given in the figure below.

Figure 8.2: A generator, a server and an exit



The model below shows an example of such a configuration. Process `G` sends a stream of integer values `0, 1, 2, 3, 4` to another process via channel `a`. Process `S` receives a value via channel `a`, assigns this value to variable `x`, doubles the value of the variable, and sends the value of the

variable via `b` to another process. Process `E` receives a value via channel `b`, assigns this value to the variable `x`, and prints this value.

After printing these five lines, process `G` stops, and processes `S` and `E` are no longer able to receive anything, so the simulation ends.

In [4]:

```
@process
def Generator(env, c_out):
    for x in range(5):
        sending = c_out.send(x)
        yield env.execute(sending)

@process
def Server(env, c_in, c_out):
    while True:
        receiving = c_in.receive()
        x = yield env.execute(receiving)

        sending = c_out.send(2*x)
        yield env.execute(sending)

@process
def Exit(env, c_in):
    while True:
        receiving = c_in.receive()
        x = yield env.execute(receiving)
        print(f"The Exit process received {x}")

def model():
    env = Environment()
    a = Channel(env)
    b = Channel(env)

    G = Generator(env, a)
    S = Server(env, a, b)
    E = Exit(env, b)

    env.run()

model()
```

executed in 105ms, finished 14:25:23 2021-11-08

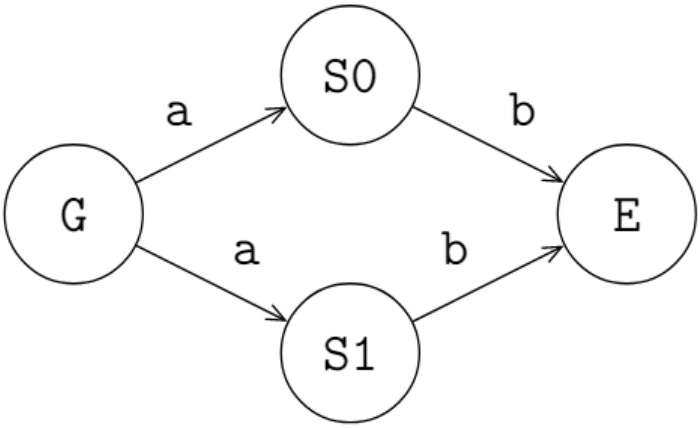
```
The Exit process received 0
The Exit process received 2
The Exit process received 4
The Exit process received 6
The Exit process received 8
```

8.3 More senders or receivers

Channels send a message (or a signal in case of synchronization channels) from one sender to one receiver. It is however allowed to give the same channel to several sender or receiver processes. The channel chooses a sender and a receiver before each communication.

The following example gives an illustration, see the figure below.

Figure 8.3: A generator, two servers and an exit



Suppose that only `G` and `S0` want to communicate. The channel wil choose a sender (namely `G`) and a receiver (process `S0`), and let both processes communicate with each other. When sender `G`, and both receivers (`S0` and `S1`), want to communicate, the channel chooses a sender (`G` as it is the only sender available to the channel), and a receiver (either process `S0` or process `S1`), and it lets the chosen processes communicate with each other. This selection process is random.

Sharing a channel in this way allows to send data to receiving processes where the receiving party is not relevant (either server process will do). This way of communication is different from *broadcasting*, where both servers receive the same data value. Broadcasting is not supported in this simulation tool.

In case of two senders, `S0` and `S1`, and one receiver `E` the selection process is the same. If one of the two servers `S` can communicate with exit `E`, communication between that server and the exit takes place. If both servers can communicate, a random choice is made. Having several senders and several receivers for a single channel is also handled in the same manner. A random choice is made for the sending process and a random choice is made for the receiving process before each communication. To communicate with several other processes but without non-determinism, unique channels must be used.

Below is the model for the configuration with two servers.

In [5]:

```
@process
def Generator(env, c_out):
    for i in range(5):
        yield env.timeout(1)
        x = f"Entity {i} passed through G"
        yield env.execute(c_out.send(x))

@process
def Server(env, c_in, c_out, s):
    while True:
        x = yield env.execute(c_in.receive())
        x = x + f", S{s}"
        yield env.execute(c_out.send(x))

@process
def Exit(env, c_in):
    while True:
        x = yield env.execute(c_in.receive())
        print(x + " and E.")

def model():
    env = Environment()
    a = Channel(env)
    b = Channel(env)

    G = Generator(env, a)
    S0 = Server(env, a, b, 0)
    S1 = Server(env, a, b, 1)

    E = Exit(env, b)

    env.run()

model()
```

executed in 8ms, finished 14:25:23 2021-11-08

```
Entity 0 passed through G, S0 and E.
Entity 1 passed through G, S0 and E.
Entity 2 passed through G, S1 and E.
Entity 3 passed through G, S1 and E.
Entity 4 passed through G, S0 and E.
```

8.4 Many channels

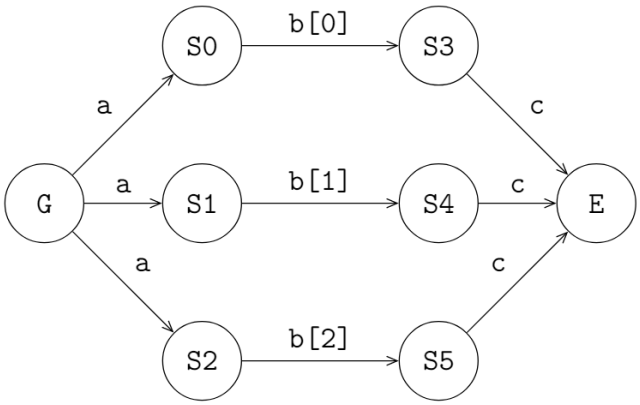
Multiple channels can be defined at once using list comprehension. This can be useful in combination with list comprehension for processes.

Below is an example of using list comprehension to quickly create three channels at once for three parallel production processes.

Typesetting math: 100%

Figure 8.4: Three parallel production processes

Figure 8.4: Three parallel production processes



The model for this example is shown below. We can re-use the process functions defined in the previous example; only the model needs to be redefined.

In [6]:

```
def model():
    N = 3 # the number of parallel processes

    env = Environment()
    a = Channel(env)
    b = [Channel(env) for i in range(N)]
    c = Channel(env)

    G = Generator(env, a)
    S012 = [Server(env, a, b[i], i) for i in range(N)]
    S345 = [Server(env, b[i], c, i+N) for i in range(N)]
    E = Exit(env, c)

    env.run()

model()
```

executed in 6ms, finished 14:25:23 2021-11-08

```
Entity 0 passed through G, S2, S5 and E.
Entity 1 passed through G, S1, S4 and E.
Entity 2 passed through G, S2, S5 and E.
Entity 3 passed through G, S0, S3 and E.
Entity 4 passed through G, S2, S5 and E.
```

8.5 Monitoring multiple communication events

In some situations it might be necessary for a process to monitor multiple communication events at once. For example, when a process wants to send one entity across either of two channels, which means we need to monitor both communication events to watch if either of them accepts the entity, and we must then execute only one of the communication events (not over both!).

We can achieve this using the `select` function: `Environment.select(communication event 1, communication event 2, ...)`. This function takes multiple communication events as input. When the select function is used, the simulation monitors if any of the communication events can occur, and then executes the first one that can. If multiple communication events are able to occur at the same time, one is *selected* at random (hence the name of the function). The communication events which were not selected are aborted. The end result is the same as if the selected communication event was executed using `Environment.execute(selected communication event)`.

This function accepts both communication events, as well as lists of communication events (lists must be preceded by an asterisk `*`). Examples are:

- Multiple communication events as input: `Environment.select(communication event 1, communication event 2, ...)`
- A list of communication events as input: `Environment.select(*a list of communication events)`
- A combination of the two as input: `Environment.select(*list of communication events 1-3, communication event 4, ...)`

The output when yielding a select function `output = yield Environment.select(...)` depends on what type of communication event was selected; the function returns the received entity as output when a receive event was selected, and it returns `None` as output when a send event was selected.

Below is an example of a provider which is able to provide across two different channels.

In [7]:

```
@process
def DualProvider(env, c_out1, c_out2):
    for x in range(10):
        sending1 = c_out1.send(x)
        sending2 = c_out2.send(x)
        yield env.select(sending1, sending2) # we try to send the entity over both

@process
def Consumer(env, c_in, i):
    while True:
        x = yield env.execute(c_in.receive())
        print(f"Consumer {i} received entity {x}")


def model():
    env = Environment()
    a = Channel(env)
    b = Channel(env)
    DP = DualProvider(env, a, b)
    C1 = Consumer(env, a, 1)
    C2 = Consumer(env, b, 2)
    env.run()

model()
```

executed in 10ms, finished 14:25:23 2021-11-08

```
Consumer 1 received entity 0
Consumer 2 received entity 1
Consumer 1 received entity 2
Consumer 2 received entity 3
Consumer 1 received entity 4
Consumer 2 received entity 5
Consumer 1 received entity 6
Consumer 2 received entity 7
Consumer 1 received entity 8
Consumer 2 received entity 9
```

8.5.1 Describing the outcomes by using the selected function

In some instances, a process will take a different action depending on which communication event was selected. We can check if a communication event was selected by using the `selected(communication event)` function, which returns `True` if the communication event has occurred. This is useful when we want to describe different outcomes depending on which event was selected.

The example below shows how we can check if either `sending1` or `sending2` has been selected. The same example also gives an example of using a list of communication events as input for the select function.

In [8]:

```
@process
def DualProvider(env, c_out1, c_out2):
    for x in range(10):
        sending1 = c_out1.send(x)
        sending2 = c_out2.send(x)
        communication_events = [sending1, sending2]
        yield env.select(*communication_events)
        if selected(sending1):
            print(f"Provider sent entity {x} across channel a")
        if selected(sending2):
            print(f"Provider sent entity {x} across channel b")

@process
def Consumer(env, c_in, i):
    while True:
        x = yield env.execute(c_in.receive())

def model():
    env = Environment()
    a = Channel(env)
    b = Channel(env)
    DP = DualProvider(env, a, b)
    C1 = Consumer(env, a, 1)
    C2 = Consumer(env, b, 2)
    C3 = Consumer(env, a, 3)
    C4 = Consumer(env, b, 4)
    env.run()

model()
```

executed in 8ms, finished 14:25:23 2021-11-08

```
Provider sent entity 0 across channel a
Provider sent entity 1 across channel a
Provider sent entity 2 across channel b
Provider sent entity 3 across channel b
Provider sent entity 4 across channel a
Provider sent entity 5 across channel b
Provider sent entity 6 across channel a
Provider sent entity 7 across channel a
Provider sent entity 8 across channel b
Provider sent entity 9 across channel b
```

8.5.2 Guards

Suppose that a process needs to watch different sets of communication events in different scenario's. We could describe every different scenario explicitely, but this would lead to bloated code. Instead, we can add a *guard* to our communication events. A guard can be very useful when a process needs to watch multiple different communication events at the same time, with some communication events only being allowed under certain conditions.

A guard is a boolean function which denotes under which conditions a communication event is allowed to take place. A sending event with a guard would look as follows:

- `sending = channel.send(entity) if guard_sending else None`
  - with `guard_sending` being a boolean expression denoting if sending is allowed to take place.

We can use guards this way because the `Environment.select(...)` function skips any communication event which has as value `None`. If there are no communication events for which the guard is `True`, then the process continues without any communication.

Below is an example of an the dual provider using guards. In this scenario:

- It only sends across channel1 if `x` is a multiple of 2.
- It only sends across channel2 if `x` is a multiple of 3.

This translates to `guard1 = (x % 2 == 0)` for `sending1` and `guard2 = (x % 3 == 0)` for `sending2`. The result is:

- If both `guard1` and `guard2` are true (e.g. `x=6`), select either `sending1` or `sending2` randomly
- If either of the guards is true, and the other is false (e.g. `x=3`), select the corresponding communication event.
- If neither of the guards is true (e.g. `x=1`), throw away entity `x`.

In [9]:

```
@process
def DualProvider(env, c_out1, c_out2):
    for x in range(1,21):
        guard1 = (x % 2 == 0)  # The guard for sending across channel c_out1: only
        guard2 = (x % 3 == 0)  # The guard for sending across channel c_out2: only

        sending1 = c_out1.send(x) if guard1 else None  # only execute sending1 when
        sending2 = c_out2.send(x) if guard2 else None  # only execute sending2 when

        communication_events = [sending1, sending2]
        yield env.select(*communication_events)
        if selected(sending1):
            print(f"Provider sent entity {x} across channel a")
        if selected(sending2):
            print(f"Provider sent entity {x} across channel b")
        if not selected(sending1) and not selected(sending2):
            print(f"Provider throws away entity {x}")

@process
def Consumer(env, c_in, i):
    while True:
        x = yield env.execute(c_in.receive())

def model():
    env = Environment()
    a = Channel(env)
    b = Channel(env)
    DP = DualProvider(env, a, b)
    C1 = Consumer(env, a, 1)
    C2 = Consumer(env, b, 2)
    env.run()

model()
```

executed in 10ms, finished 14:25:23 2021-11-08

```
Provider throws away entity 1
Provider sent entity 2 across channel a
Provider sent entity 3 across channel b
Provider sent entity 4 across channel a
Provider throws away entity 5
Provider sent entity 6 across channel a
Provider throws away entity 7
Provider sent entity 8 across channel a
Provider sent entity 9 across channel b
Provider sent entity 10 across channel a
Provider throws away entity 11
Provider sent entity 12 across channel a
Provider throws away entity 13
Provider sent entity 14 across channel a
Provider sent entity 15 across channel b
Provider sent entity 16 across channel a
Provider throws away entity 17
Provider sent entity 18 across channel a
```

```
Provider throws away entity 19
Provider sent entity 20 across channel a
```

8.6 Summary

- A channel is defined with `channel = Channel(environment)`
- Communication over a channel is done using communication events.
  - A process can send an entity over a channel with communication event `sending = channel.send(entity)`
  - A process can receive over a channel with communication event `receiving = channel.receive()`
  - Communication events do not start automatically, they need to be executed through `env.execute(communication event)`
- A channel can transmit any type of entity; for example integers, strings, or even a custom data type
  - a Signal without data is named a *synchronization signal* and is done through: `channel.send(None)`
- A channel can have several processes sending and/or receiving.
  - If multiple processes want to send/receive at the same time, then the channel will select a sender and a receiver at random.
- Multiple channels can be defined using list comprehension, which can be useful in combination with list comprehension for processes.
  - An example would be `channels = [Channel(env) for i in range(10)]`
- Monitoring multiple channels
  - A process can select one of multiple communication events. Only the first communication event which is able to communicate will be executed, the others are aborted. If multiple communication events are able to communicate at the same time, then one is chosen at random. This can be done through:
    - `env.select(communication event 1, communication event 2, ...)`
    - `env.select(*list of communication events)`
    - or with a combination of the two: `env.select(*list of communication events 1-3, communication event 4, ...)`
  - We can check if a communication event was selected by using the `selected(communication event)` function, which returns `True` if the communication event has occurred. This is useful when we want to describe different outcomes depending on which event was selected.
  - Guards are boolean functions which denotes under which conditions a communication event is allowed to take place.
    - An example of a send event with a guard would be: `sending = channel.send(entity) if guard_sending else None`
      - with `guard_sending` being a boolean expression denoting if sending is allowed to take place.

8.7 Exercises

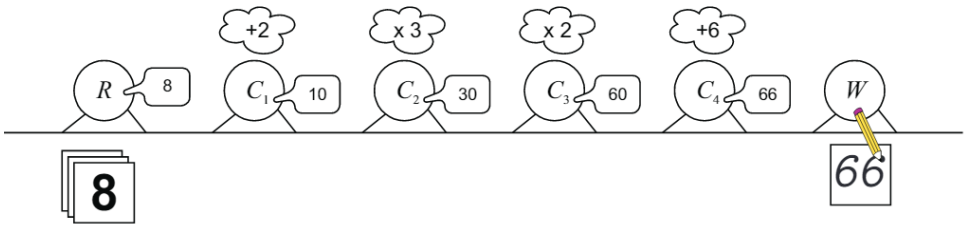
1. Given is the specification of process `P` and model `M`. Why does the model terminate immediately?

In [ ]:

```
@process
def P(env, c_in, c_out):
    x = 0
    while True:
        x = yield env.execute(c_in.receive())
        x = x + 1
        print(x)
        yield env.execute(c_out.send())

def M():
    env = Environment()
    a = Channel(env)
    b = Channel(env)

    P1 = P(a,b)
    P2 = P(b,a)

    env.run()
```

2. Six children have been given the assignment to perform a series of calculations on the numbers $0, 1, 2, 3, \ldots, 9$, namely add 2, multiply by 3, multiply by 2, and add 6 subsequently. They decide to split up the calculations and to operate in parallel. They sit down at a table next to each other. The first child, the reader R, reads the numbers $0, 1, 2, 3, \ldots, 9$ one by one to the first calculating child C_1. Child C_1 adds 2 and tells the result to its right neighbour, child C_2. After telling the result to child C_2, child C_1 is able to start calculating on the next number the reader R tells him. Children C_2, C_3, and C_4 are analogous to child C_1; they each perform a different calculation on a number they hear and tell the result to their right neighbour. At the end of the table the writer W writes every result he hears down on paper. The figure below shows a schematic drawing of the children at the table.

Figure 8.5: The six children



a. Finish the specification for the reading child R, that reads the numbers 0 till 9 one by one.

In [ ]:

```
@process
def R(env, ...):
    i = 0
    while i < 10:
        ...
```

b. Specify the parameterized process $C_{*add*}*thatrepresentsthechildren*C_1*and*C_4$, who perform an addition.

In [ ]:

```
@process
def C_add(env, ...):
    while True:
        ...
```

c. Specify the parameterized process $C_{*mul*}*thatrepresentsthechildren*C_2*and*C_3$, who perform a multiplication.

In [ ]:

```
@process
def C_mul(env, ...):
    while True:
        ...
```

d. Specify the process W representing the writing child. Print each result as output.

In [ ]:

```
@process
def W(env, a):
    while True:
        ...
```

e. Make a graphical representation of the model `SixChildren` that is composed of the six children.

f. Specify the model `SixChildren` . Simulate the model.

In [ ]:

```
def SixChildren():
    env = Environment()

    ...

    env.run()

SixChildren()
```

8.7 Answers to exercises

[Click for the answer to a]
[Click for the answer to b]
[Click for the answer to c]

[Click for the answer to d]
[Click for the answer to e]
[Click for the answer to f]

In [ ]:

To use the examples in this chapter, first run the code below to import the right libraries.

In [1]:

```
# ================================
# Imports
# ================================
from PyCh import *
from numpy import random
from dataclasses import dataclass
import math
```

executed in 369ms, finished 14:25:43 2021-11-08

```
PyCh version 1.0 imported succesfully.
```

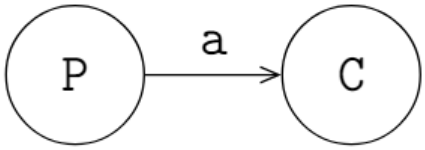
9 Buffers

In the previous chapter we introduced channels. Using channels, entities can be transfered from one process to the next, in a synchronous manner (Sender and receiver perform the communication at exactly the same moment in time, and the communication is instantaneous). In many systems however, processes do not use synchronous communication, they use asynchronous communication instead. Entities (products, packets, messages, simple tokens, and so on) are sent, temporarily stored in a *buffer*, and then received.

In fact, the decoupling of sending and receiving is very important, it allows compensating temporarily differences between the number of items that are sent and received (Under the assumption that the receiver is fast enough to keep up with the sender in general, otherwise the buffer will grow forever or overflow).

For example, consider the exchange of items from a producer process `P` to a consumer process `C` as shown in the figure below. In the unbuffered situation, both processes communicate at the same time. This means that when one process is (temporarily) faster than the other, it has to wait for the other process before communication can take place.

Figure 9.1: A producer and a consumer



As shown in the example below, when the consumer needs more time to process entities than the producer, then the producer will have to wait for the consumer.

In [2]:

```
@process
def producer(env, c_out):
    total_t_waiting = 0
    for i in range(5):
        t_ready = env.now
        yield env.execute(c_out.send(i))
        t_waiting = env.now - t_ready
        total_t_waiting = total_t_waiting + t_waiting
        print(f"The producer has sent entity {i} at t = {env.now:.2f}, "
              f"and had to wait {t_waiting} seconds. Total waiting time so far is {
        yield env.timeout(1)

@process
def consumer(env, c_in):
    while True:
        i = yield env.execute(c_in.receive())
        yield env.timeout(3)

def model():
    env = Environment()
    a = Channel(env)
    P = producer(env, a)
    C = consumer(env, a)
    env.run()

model()
```

executed in 54ms, finished 14:25:43 2021-11-08

```
The producer has sent entity 0 at t = 0.00, and had to wait 0 seconds.
Total waiting time so far is 0 seconds.
The producer has sent entity 1 at t = 3.00, and had to wait 2.0 second
s. Total waiting time so far is 2.0 seconds.
The producer has sent entity 2 at t = 6.00, and had to wait 2.0 second
s. Total waiting time so far is 4.0 seconds.
The producer has sent entity 3 at t = 9.00, and had to wait 2.0 second
s. Total waiting time so far is 6.0 seconds.
The producer has sent entity 4 at t = 12.00, and had to wait 2.0 second
s. Total waiting time so far is 8.0 seconds.
```

With a buffer in-between, the producer can give its item to the buffer, and continue with its work. Likewise, the consumer can pick up a new item from the buffer at any later time (if the buffer has items). In this simulation tool, buffers are not modeled as channels, they are modeled as additional processes instead. The result is shown in Figure 9.2 below.

Figure 9.2: A producer and a consumer, with an additional buffer process.

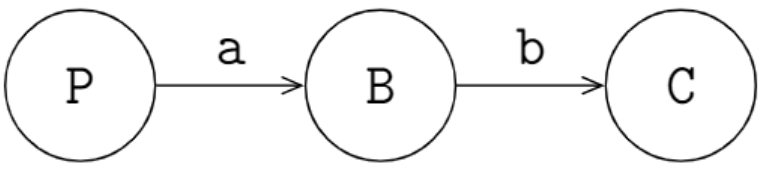


Figure 9.2: A producer and a consumer, with an additional buffer process.

The producer sends its items synchronously (using channel `a`) to the buffer process. The buffer process keeps the item until it is needed. The consumer gets an item synchronously (using channel `b`) from the buffer when it needs a new item (and one is available).

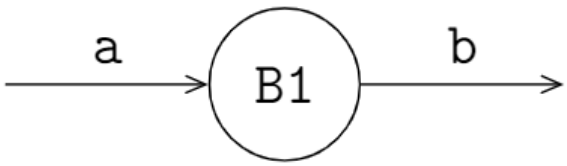
In manufacturing networks, buffers, in combination with servers, play a prominent role, for buffering items in the network. Various buffer types exist in these networks: buffers can have a finite or infinite capacity, they have an input/output discipline, for example a first-out queuing discipline or a priority-based discipline. Buffers can store different kinds of items, for example, product-items, information-items, or a combination of both. Buffers may also have sorting facilities, etc.

In this chapter some buffer types are described, and with the presented concepts numerous types of buffer can be designed by the engineer. First a simple buffer process with one buffer position is presented, followed by more advanced buffer models. The producer and consumer processes are not discussed in this chapter.

9.1 A one-place buffer

A buffer usually has a receiving channel and a sending channel, for receiving and sending items. A buffer, buffer `B1`, is presented in Figure 9.3.

Figure 9.3: A 1-place buffer.



The simplest buffer is a one-place buffer, for buffering precisely one item. A one-place buffer is shown below. `c_in` and `c_out` are the receiving and sending channels. Entity `i` is buffered in the process. A buffer receives an item, stores the item, and sends the item to the next process, if the next process is willing to receive the item. The buffer is not willing to receive a second item, as long as the first item is still in the buffer.

In [3]:

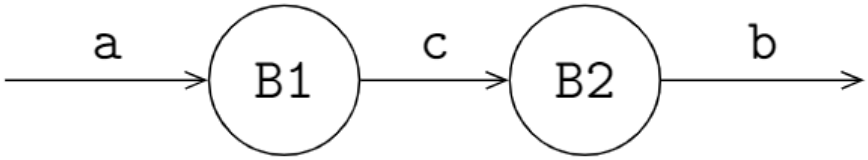
```
@process
def buffer(env, c_in, c_out):
    while True:
        i = yield env.execute(c_in.receive())
        yield env.execute(c_out.send(i))

def model():
    env = Environment()
    a = Channel(env)
    b = Channel(env)
    P = producer(env, a)
    B1 = buffer(env, a, b)
    C = consumer(env, b)
    env.run()

model()
```

executed in 9ms, finished 14:25:43 2021-11-08

```
The producer has sent entity 0 at t = 0.00, and had to wait 0 seconds.
Total waiting time so far is 0 seconds.
The producer has sent entity 1 at t = 1.00, and had to wait 0 seconds.
Total waiting time so far is 0 seconds.
The producer has sent entity 2 at t = 3.00, and had to wait 1.0 second
s. Total waiting time so far is 1.0 seconds.
The producer has sent entity 3 at t = 6.00, and had to wait 2.0 second
s. Total waiting time so far is 3.0 seconds.
The producer has sent entity 4 at t = 9.00, and had to wait 2.0 second
s. Total waiting time so far is 5.0 seconds.
```

A two-place buffer can be created, by using the one-place buffer process twice. A two-place buffer is depicted below.

Figure 9.4: A 2-place buffer.



A two-place buffer is shown below. In the two-place buffer, processes `B1` and `B2` buffer maximal two items. If each buffer process contains an item, a third item has to wait in front of process `B1` .

Note that `buffer_2_place` is not a process but a submodel, because it does not yield any events by itself

In [4]:

```
def buffer_2_place(env, c_in, c_out):
    c_B1B2 = Channel(env)
    B1 = buffer(env, c_in, c_B1B2)
    B2 = buffer(env, c_B1B2, c_out)

def model():
    env = Environment()
    a = Channel(env)
    b = Channel(env)
    P = producer(env, a)
    B = buffer_2_place(env, a, b)
    C = consumer(env, b)
    env.run()

model()
```

executed in 6ms, finished 14:25:43 2021-11-08

```
The producer has sent entity 0 at t = 0.00, and had to wait 0 seconds.
Total waiting time so far is 0 seconds.
The producer has sent entity 1 at t = 1.00, and had to wait 0 seconds.
Total waiting time so far is 0 seconds.
The producer has sent entity 2 at t = 2.00, and had to wait 0 seconds.
Total waiting time so far is 0 seconds.
The producer has sent entity 3 at t = 3.00, and had to wait 0.0 second
s. Total waiting time so far is 0.0 seconds.
The producer has sent entity 4 at t = 6.00, and had to wait 2.0 second
s. Total waiting time so far is 2.0 seconds.
```

This procedure can be extended to create even larger buffers. Another, more preferable manner however, is to describe a buffer in a single process by using a `select` statement and a list for storage of the items. Such a buffer is discussed in the next section.

9.2 A single process buffer

An informal description of the process of a buffer, with an arbitrary number of stored items, is the following:

- If the buffer has space for an item, *and* can receive an item from another process via channel `a`, the buffer process receives that item, and stores the item in the buffer.
- If the buffer contains at least one item, *and* the buffer can send that item to another process via channel `b`, the buffer process sends that item, and removes that item from the buffer.
- If the buffer can both send and receive a value, the buffer process selects one of the two possibilities (in a non-deterministic manner).
- If the buffer can not receive an item, and can not send an item, the buffer process waits.

Using the select function we learned in the previous chapter, we can create a buffer process, which can simultaneously send and receive items. An example of a finite buffer with `N` capacity is shown below. In the model of the buffer, list `xs` is used for storing the received items.

This buffer can only send when there are items in the buffers (the guard for sending is `len(xs)>0`), and can only receive when the buffer is not yet full (the guard for receiving is `len(xs)<N`). The buffer then either sends or receives an item:

```
x = yield env.select(sending, receiving)
```

Next to the sending and receiving of items (to and from the buffer process) is the question of how to order the stored items. A common form is the *first-in first-out* (fifo) queuing discipline. Items that enter the buffer first (first-in) also leave first (first-out), the order of items is preserved by the buffer process. Our buffer is has such a fifo policy.

If an item `x` is receive, it is added at the rear of list:

```
if selected(receiving):
        xs = xs + [x]
```

If an item `x` is receive, it is removed from the start of the list:

```
if selected(sending):
        xs = xs[1:]
```

The model of the finite buffer is shown below:

In [5]:

```
# Finite buffer model
@process
def buffer(env, c_in, c_out, N):
    xs = []
    while True:
        sending = c_out.send(xs[0]) if len(xs)>0 else None  # If there is an item i
        receiving = c_in.receive()  if len(xs)<N else None  # If the buffer is not
        x = yield env.select(sending, receiving)
        if selected(receiving):
            xs = xs + [x]
        if selected(sending):
            xs = xs[1:]
        print(f"The buffer contains {len(xs)} item(s)")
```

executed in 5ms, finished 14:25:43 2021-11-08

In [6]:

```
# Rest of model
@process
def producer(env, c_out):
    for x in range(5):
        yield env.execute(c_out.send(x))
        yield env.timeout(1)

@process
def consumer(env, c_in):
    while True:
        x = yield env.execute(c_in.receive())
        yield env.timeout(3)

def model():
    env = Environment()
    a = Channel(env)
    b = Channel(env)
    P = producer(env, a)
    B = buffer(env, a, b, 3)
    C = consumer(env, b)
    env.run()

model()
```

executed in 8ms, finished 14:25:43 2021-11-08

```
The buffer contains 1 item(s)
The buffer contains 0 item(s)
The buffer contains 1 item(s)
The buffer contains 2 item(s)
The buffer contains 1 item(s)
The buffer contains 2 item(s)
The buffer contains 3 item(s)
The buffer contains 2 item(s)
The buffer contains 1 item(s)
The buffer contains 0 item(s)
```

Similarly, a buffer with an infinite capacity can be written as following:

In [7]:

```
# Infinite buffer model (FIFO)
@process
def buffer(env, c_in, c_out):
    xs = []
    while True:
        sending = c_out.send(xs[0]) if len(xs)>0 else None
        receiving = c_in.receive()
        x = yield env.select(sending, receiving)
        if selected(receiving):
            xs = xs + [x]
        if selected(sending):
            xs = xs[1:]
        print(f"The buffer contains {len(xs)} item(s)")
```

executed in 5ms, finished 14:25:43 2021-11-08

In [8]:

```
# Rest of model
@process
def producer(env, c_out):
    for x in range(5):
        yield env.execute(c_out.send(x))
        yield env.timeout(1)

@process
def consumer(env, c_in):
    while True:
        x = yield env.execute(c_in.receive())
        yield env.timeout(3)

def model():
    env = Environment()
    a = Channel(env)
    b = Channel(env)
    P = producer(env, a)
    B = buffer(env, a, b)
    C = consumer(env, b)
    env.run()

model()
```

executed in 14ms, finished 14:25:43 2021-11-08

```
The buffer contains 1 item(s)
The buffer contains 0 item(s)
The buffer contains 1 item(s)
The buffer contains 2 item(s)
The buffer contains 1 item(s)
The buffer contains 2 item(s)
The buffer contains 3 item(s)
The buffer contains 2 item(s)
The buffer contains 1 item(s)
The buffer contains 0 item(s)
```

A first-in first-out buffer is also called a *queue*, while a first-in last-out (*lifo*) buffer is called a *stack*. A lifo buffer puts the last received item at the head of the list, and gets the first item from the list.

```
    if selected(receiving):
            xs = [x] + xs
```

A model of a lifo buffer is:

In [9]:

```
# Lifo buffer model
@process
def buffer(env, c_in, c_out):
    xs = []
    while True:
        sending = c_out.send(xs[0]) if len(xs)>0 else None
        receiving = c_in.receive()
        x = yield env.select(sending, receiving)
        if selected(receiving):
            xs = [x] + xs   # items are placed on top of the stack
        if selected(sending):
            xs = xs[1:]
        print(f"The buffer contains {len(xs)} item(s)")
```

executed in 6ms, finished 14:25:43 2021-11-08

In [10]:

```
# Rest of model
@process
def producer(env, c_out):
    for x in range(5):
        yield env.execute(c_out.send(x))
        yield env.timeout(1)

@process
def consumer(env, c_in):
    while True:
        x = yield env.execute(c_in.receive())
        yield env.timeout(3)

def model():
    env = Environment()
    a = Channel(env)
    b = Channel(env)
    P = producer(env, a)
    B = buffer(env, a, b)
    C = consumer(env, b)
    env.run()

model()
```

executed in 8ms, finished 14:25:43 2021-11-08

```
The buffer contains 1 item(s)
The buffer contains 0 item(s)
The buffer contains 1 item(s)
The buffer contains 2 item(s)
The buffer contains 1 item(s)
The buffer contains 2 item(s)
The buffer contains 3 item(s)
The buffer contains 2 item(s)
The buffer contains 1 item(s)
The buffer contains 0 item(s)
```

9.3 A token buffer

In the next example, signals are buffered instead of items. The buffer receives and sends 'empty' items or *tokens*. Counter variable `w` denotes the difference of the number of tokens received and the number of tokens sent. If the buffer receives a token, counter `w` is incremented; if the buffer sends a token, counter `w` is decremented. If the number of tokens sent is less than the number of tokens received, there are tokens in the buffer, and `w > 0`. A receiving channel `a` is defined for receiving tokens. A sending channel `b` is defined for sending tokens. The buffer becomes:

In [11]:

```
# Token buffer model
@process
def buffer(env, c_in, c_out):
    w = 0;
    while True:
        sending = c_out.send() if w>0 else None
        receiving = c_in.receive()
        yield env.select(sending, receiving)
        if selected(receiving):
            w = w + 1
        if selected(sending):
            w = w - 1
        print(f"The buffer contains {w} token(s)")
```

executed in 4ms, finished 14:25:43 2021-11-08

In [12]:

```
# Rest of model
@process
def producer(env, c_out):
    for i in range(5):
        yield env.execute(c_out.send())
        yield env.timeout(1)

@process
def consumer(env, c_in):
    while True:
        yield env.execute(c_in.receive())
        yield env.timeout(3)


def model():
    env = Environment()
    a = Channel(env)
    b = Channel(env)
    P = producer(env, a)
    B = buffer(env, a, b)
    C = consumer(env, b)
    env.run()

model()
```

executed in 20ms, finished 14:25:43 2021-11-08

```
The buffer contains 1 token(s)
The buffer contains 0 token(s)
The buffer contains 1 token(s)
The buffer contains 2 token(s)
The buffer contains 1 token(s)
The buffer contains 2 token(s)
The buffer contains 3 token(s)
The buffer contains 2 token(s)
The buffer contains 1 token(s)
The buffer contains 0 token(s)
```

9.4 A priority buffer

A buffer for items with different priority is described in this section. An item has a high priority or a normal priority. Items with a high priority should leave the buffer first.

In this example, an item is a dataclass with a field `prio`, denoting the priority, `0` for high priority, and `1` for normal priority:

In [13]:

```
@dataclass
class Item:
    prio: int = 0
```

executed in 3ms, finished 14:25:43 2021-11-08

In the model the received items are, on the basis of the value of the `prio` -field in the item, stored in one of two lists: one list for 'high' items and one list for 'normal' items. The discipline of the buffer is that items with a high priority leave the buffer first. For the storage of items, two lists are used: a list for high priority items and a list for normal priority items. The two lists are described by a list of two lists `xs` :

```
xs = [ [], [] ];
```

List `xs[0]` contains the high priority items, `xs[1]` the normal priority items. The behavior of the priority buffer is as following:

- Received items `x` are stored in `xs[x.prio]` by the statement `xs[x.prio] = xs[x.prio] + [x]` .
- If the list of high priority items is not empty (`len(xs[0])>0`), items with high priority are sent. The first element in list `xs[0]` is element `xs[0][0]` .
- If there are no high priority items (`len(xs[0])==0`), and there are normal priority items (`len(xs[1])>0`), the first element of list `xs[1]` , element `xs[1][0]` , is sent.

The model for the priority buffer is shown below:

In [14]:

```
# Priority buffer model
@process
def buffer(env, c_in, c_out):
    xs = [ [], [] ];
    while True:
        sending_prio   = c_out.send(xs[0][0]) if len(xs[0])>0 else None
        sending_normal = c_out.send(xs[1][0]) if (len(xs[1])>0 and len(xs[0])==0) e
        receiving = c_in.receive()
        x = yield env.select(sending_prio, sending_normal, receiving)
        if selected(receiving):
            xs[x.prio] = xs[x.prio] + [x]
        if selected(sending_prio):
            xs[0] = xs[0][1:]
        if selected(sending_normal):
            xs[1] = xs[1][1:]
        print(f"The buffer contains {len(xs[1])} normal item(s) and {len(xs[0])} pr
```

executed in 8ms, finished 14:25:43 2021-11-08

In [15]:

```
# Rest of model
@dataclass
class Item:
    prio: int = 0

@process
def producer(env, c_out):
    random_prio = lambda: random.randint(2)  # priority is assigned randomly
    for i in range(10):
        x = Item(prio = random_prio())
        yield env.execute(c_out.send(x))
        yield env.timeout(1)

@process
def consumer(env, c_in):
    while True:
        x = yield env.execute(c_in.receive())
        yield env.timeout(3)


def model():
    env = Environment()
    a = Channel(env)
    b = Channel(env)
    P = producer(env, a)
    B = buffer(env, a, b)
    C = consumer(env, b)
    env.run()

model()
```

executed in 13ms, finished 14:25:43 2021-11-08

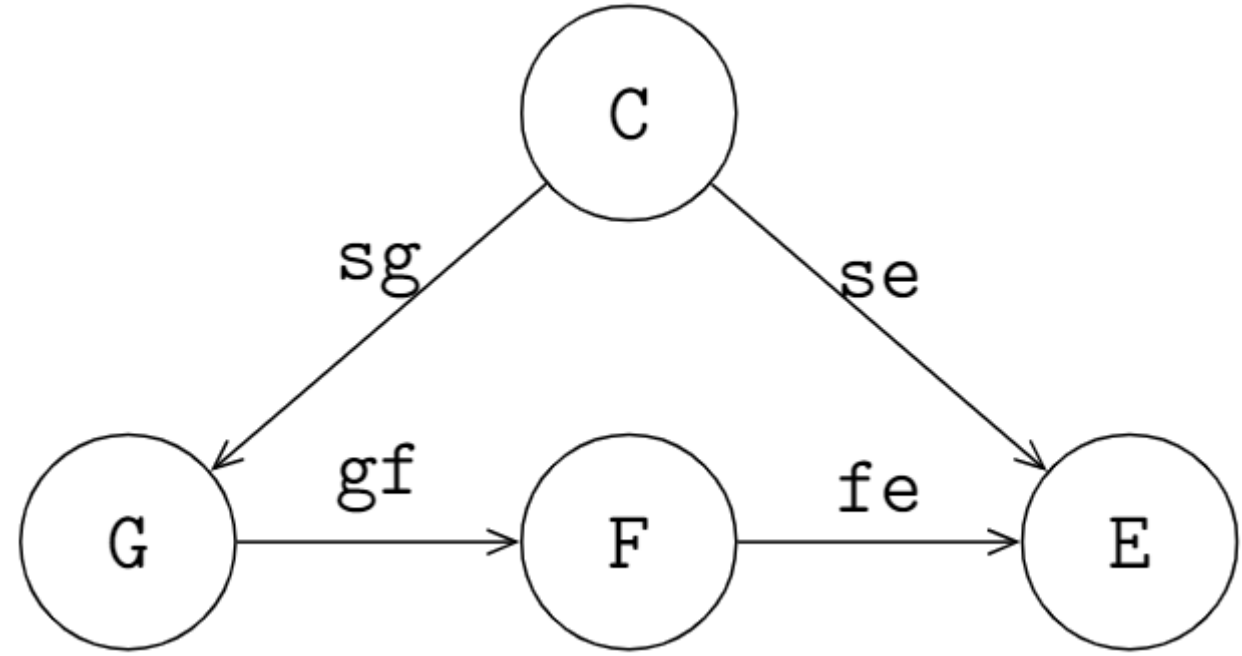
```
The buffer contains 0 normal item(s) and 1 priority items
The buffer contains 0 normal item(s) and 0 priority items
The buffer contains 0 normal item(s) and 1 priority items
The buffer contains 0 normal item(s) and 2 priority items
The buffer contains 0 normal item(s) and 1 priority items
The buffer contains 1 normal item(s) and 1 priority items
The buffer contains 2 normal item(s) and 1 priority items
The buffer contains 2 normal item(s) and 2 priority items
The buffer contains 2 normal item(s) and 1 priority items
The buffer contains 2 normal item(s) and 2 priority items
The buffer contains 3 normal item(s) and 2 priority items
The buffer contains 4 normal item(s) and 2 priority items
The buffer contains 4 normal item(s) and 1 priority items
The buffer contains 5 normal item(s) and 1 priority items
The buffer contains 5 normal item(s) and 0 priority items
The buffer contains 4 normal item(s) and 0 priority items
The buffer contains 3 normal item(s) and 0 priority items
The buffer contains 2 normal item(s) and 0 priority items
The buffer contains 1 normal item(s) and 0 priority items
The buffer contains 0 normal item(s) and 0 priority items
```

9.5 Exercises

Exercise 9.5.1

To study product flow to and from a factory, a setup as shown in Figure 9.5 is created. `F` is the factory being studied, generator `G` sends products into the factory, and exit process `E` retrieves finished products. The factory is tightly controlled by controller `C` that sends a signal to `G` or `E` before a product may be moved.

Figure 9.5: A controlled factory.



The model of the system (excluding the factory model) is as follows:

In [ ]:

```
@process
def Generator(env, c_out, c_signal, N):
    for i in range(N):
        yield env.execute(c_signal.receive())
        yield env.execute(c_out.send(i))

@process
def Exit(env, c_in, c_signal):
    while True:
        yield env.execute(c_signal.receive())
        x = yield env.execute(c_in.receive())
        print(f"The Exit received {x}")

@process
def Controller(env, c_signal_gen, c_signal_exit, low, high):
    count = 0
    while True:
        while count < high:
            yield env.execute(c_signal_gen.send())
            count = count+1
        while count > low:
            yield env.execute(c_signal_exit.send())
            count = count-1

def model(low, high, N):
    env = Environment()
    sg = Channel(env)
    se = Channel(env)
    gf = Channel(env)
    fe = Channel(env)
    G = Generator(env, gf, sg, N)
    F = Factory(env, gf, fe)
    E = Exit(env, fe, se)
    C = Controller(env, sg, se, low, high)
    env.run()
```

A. As a model of the factory, use a FIFO buffer process. Run the simulation, and check whether all products are received by the exit process.

In [ ]:

```
# 1a: FIFO factory
@process
def Factory(env, c_in, c_out):
    ...
```

In [ ]:

```
# 1a: execute the model
model(low=0, high=1, N=10)
```

B. Change the control policy to `low = 1` and `high = 4`. Predict the outcome, and verify with simulation.

Predicted outcome: ...

In [ ]:

```
# 1b: execute the model
model(low=1, high=4, N=10)
```

C. The employees of the factory propose to stack the products in the factory to reduce the amount of space needed for buffering. Replace the factory process with a LIFO buffer process, run the experiments again, first with `low = 0` and `high = 1` and then with `low = 1` and `high = 4`.

In [ ]:

```
# 1a: LIFO factory
@process
def Factory(env, c_in, c_out):
    ...
```

In [ ]:

```
# 1c: execute the model
model(low=0, high=1, N=10)
```

In [ ]:

```
# 1c: execute the model
model(low=1, high=4, N=10)
```

D. You will notice that some products stay in the factory forever. Why does that happen? How should the policy be changed to ensure all products eventually leave the factory?

Answer: ...

In [ ]:

```
model(....)
```

9.6 Answers to exercises

Answer to 9.5.1

[Click for the answer to 9.5.1]

To use the examples in this chapter, first run the code below to import the right libraries.

In [1]:

```
# ================================
# Imports
# ================================
from PyCh import *
from numpy import random
from dataclasses import dataclass
import math
```

executed in 362ms, finished 14:25:51 2021-11-08

```
PyCh version 1.0 imported succesfully.
```

10 Production lines

A production line contains machines and/or persons that perform a sequence of tasks, where each machine or person is responsible for a single task. The term *server* is used for a machine or a person that performs a task. Usually the execution of a task takes time, e.g. a drilling process, a welding process, the set-up of a machine. Timing is an important aspect of a model, as it allows answering questions about time, often performance questions ('how many products can I make in this situation?').

In this chapter we discuss how to model a production line with various types of servers.

10.1 A simple production line

The first case is a small production line with a deterministic server (its task takes a fixed amount of time), while the second case uses stochastic arrivals (the moment of arrival of new items varies), and a stochastic server instead (the duration of the task varies each time).

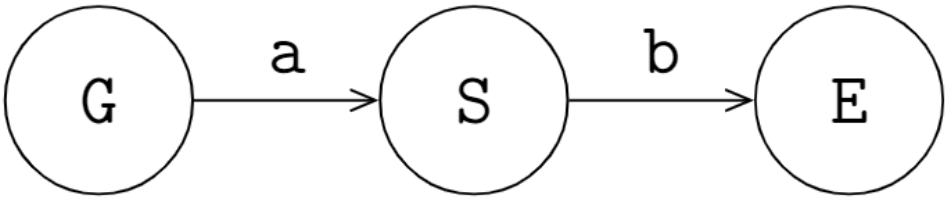
In both cases, the question is what the flow time of an item is (the amount of time that a single item is in the system), and what the throughput of the entire system is (the number of items the production line can manufacture per time unit).

10.1.1 A deterministic system

The model of a deterministic system consists of a deterministic generator, a deterministic server, and an exit process. The production line is depicted in Figure 10.1.

Figure 10.1: Generator `G`, server `S`, and exit `E`.

Figure 10.1: Generator `G`, server `S`, and exit `E`.



Generator process `G` sends items, with constant inter-arrival time `ta`, via channel `a`, to server process `S`. The server processes items with constant processing time `ts`, and sends items, via channel `b`, to exit process `E`.

An item contains a real value, denoting the creation time of the item, for calculating the throughput of the system and flow time (or sojourn time) of an item in the system. The generator process creates an item (and sets its creation time), the exit process `E` writes the measurements (the moment in time when the item arrives in the exit process, and its creation time) to the output. From these measurements, throughput and flow time can be calculated.

Model `M` describes the system:

In [2]:

```
def M(ta, ts, N):
    env = Environment()
    a = Channel(env)
    b = Channel(env)
    G = Generator(env, a, ta)
    S = Server(env, a, b, ts)
    E = Exit(env, b, N)
    env.run(until=E)
```

executed in 3ms, finished 14:25:51 2021-11-08

Parameter `ta` denotes the inter-arrival time, and is used in generator `G`. Parameter `ts` denotes the server processing time, and is used in server `S`. Parameter `N` denotes the number of items that must flow through the system to get a good measurement.

Generator process definition `Generator` has two parameters, channel `c_out`, and inter-arrival time `ta`. The description of process `G` is given by:

In [3]:

```
@process
def Generator(env, c_out, ta):
    while True:
        x = env.now
        yield env.execute(c_out.send(x))
        yield env.timeout(ta)
```

executed in 4ms, finished 14:25:51 2021-11-08

Process `G` sends an item, with the current time, and delays for `ta`, before sending the next item to server process `S`.

Server process definition `Server` has three parameters, receiving channel `c_in`, sending channel `c_out`, and server processing time `ts`:

In [4]:

```
@process
def Server(env, c_in, c_out, ts):
    while True:
        x = yield env.execute(c_in.receive())
        yield env.timeout(ts)
        yield env.execute(c_out.send(x))
```

executed in 8ms, finished 14:25:51 2021-11-08

The process receives an item from process `G`, processes the item during `ts` time units, and sends the item to exit process `E`.

Exit process definition `Exit` has two parameters, receiving channel `c_in` and the length of the experiment `N`.

In [5]:

```
@process
def Exit(env, c_in, N):
    for i in range(N):
        x = yield env.execute(c_in.receive())
        print(f"The Exit process received an item at t = {env.now:.1f} with flow ti
```

executed in 48ms, finished 14:25:51 2021-11-08

The process writes current time `env.now` and item flow time `env.now - x` to the screen for each received item. Analysis of the measurements will show that the system throughput equals $1/\mathtt{ta}$, and that the item flow time equals `ts` (if `ta` $\ge$ `ts`).

In [6]:

```
# Run the model
M(3, 1, 10)
```

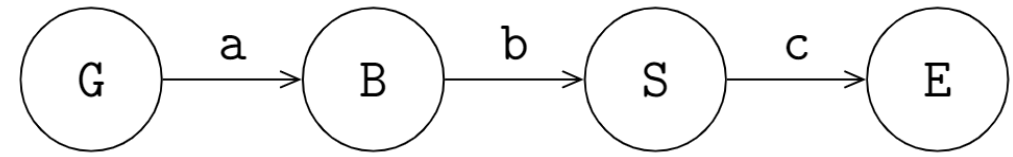
executed in 6ms, finished 14:25:51 2021-11-08

```
The Exit process received an item at t = 1.0 with flow time 1.0
The Exit process received an item at t = 4.0 with flow time 1.0
The Exit process received an item at t = 7.0 with flow time 1.0
The Exit process received an item at t = 10.0 with flow time 1.0
The Exit process received an item at t = 13.0 with flow time 1.0
The Exit process received an item at t = 16.0 with flow time 1.0
The Exit process received an item at t = 19.0 with flow time 1.0
The Exit process received an item at t = 22.0 with flow time 1.0
The Exit process received an item at t = 25.0 with flow time 1.0
The Exit process received an item at t = 28.0 with flow time 1.0
```

10.1.2 A stochastic system

In the next model, the generator produces items with an exponential inter-arrival time, and the server processes items with an exponential server processing time. To compensate for the variations in time of the generator and the server, a buffer process has been added. The model is depicted in Figure 10.2.

Figure 10.2: Generator `G`, Buffer `B`, server `S`, and exit `E`.



The model for the stochastic system runs the additional buffer process:

In [7]:

```
def StochasticSystemModel(ta, ts, N):
    env = Environment()
    a = Channel(env)
    b = Channel(env)
    c = Channel(env)
    G = Generator(env, a, ta)
    B = Buffer(env, a, b)
    S = Server(env, b, c, ts)
    E = Exit(env, c, N)
    env.run(until=E)
```

executed in 8ms, finished 14:25:51 2021-11-08

Generator `G` has two parameters, channel variable `a`, and variable `ta`, denoting the mean inter-arrival time. An `exponential` distribution is used for deciding the inter-arrival time of new items, which we sample from using `u = lambda: random.exponential(ta)`. The process sends a new item to the buffer, and then is delayed for a sample of `delay = u()` time units until the next arrival.

In [8]:

```
@process
def Generator(env, c_out, ta):
    u = lambda: random.exponential(ta)
    while True:
        x = env.now
        yield env.execute(c_out.send(x))
        delay = u()
        yield env.timeout(delay)
```

executed in 4ms, finished 14:25:51 2021-11-08

Buffer process `B` is a fifo buffer with infinite capacity, as described in chapter 9.

In [9]:

```
@process
def Buffer(env, c_in, c_out):
    xs = []
    while True:
        sending = c_out.send(xs[0]) if len(xs)>0 else None
        receiving = c_in.receive()
        x = yield env.select(sending, receiving)
        if selected(receiving):
            xs = xs + [x]
        if selected(sending):
            xs = xs[1:]
```

executed in 9ms, finished 14:25:51 2021-11-08

Server `S` has three parameters, channel variables `a` and `b`, for receiving and sending items, and a variable for the average processing time `ts`. An `exponential` distribution is used for deciding the processing time. The process receives an item from process `B`, processes the item with the sampled processing time, and sends the item to exit process `E`.

In [10]:

```
@process
def Server(env, c_in, c_out, ts):
    u = lambda: random.exponential(ts)
    while True:
        x = yield env.execute(c_in.receive())
        delay = u()
        yield env.timeout(delay)
        yield env.execute(c_out.send(x))
```

executed in 4ms, finished 14:25:51 2021-11-08

Exit process `E` is the same as in the previous case. In this case the throughput of the system also equals $1 / \mathtt{ta}$, and the *mean flow* can be obtained by doing an experiment and analysis of the resulting measurements (for `ta $>$ ts`).

In [11]:

```
# Run the model
StochasticSystemModel(3, 1, 10)
```

executed in 14ms, finished 14:25:51 2021-11-08

```
The Exit process received an item at t = 2.2 with flow time 2.2
The Exit process received an item at t = 3.3 with flow time 0.1
The Exit process received an item at t = 6.7 with flow time 1.5
The Exit process received an item at t = 8.7 with flow time 0.5
The Exit process received an item at t = 10.1 with flow time 1.2
The Exit process received an item at t = 12.8 with flow time 3.4
The Exit process received an item at t = 15.7 with flow time 4.6
The Exit process received an item at t = 16.2 with flow time 2.2
The Exit process received an item at t = 18.5 with flow time 4.3
The Exit process received an item at t = 19.0 with flow time 3.8
```

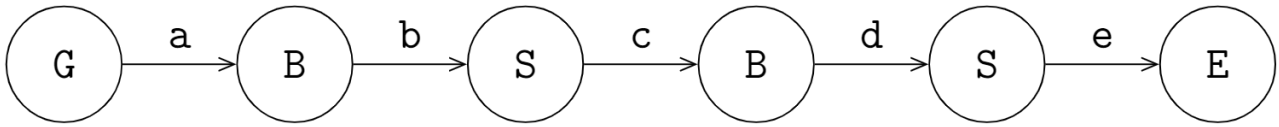
10.2 Parallel and serial processing

In this section two different types of systems are shown: a serial and a parallel system. In a serial system the servers are positioned after each other, in a parallel system the servers are operating in parallel. Both systems use a stochastic generator, and stochastic servers.

10.2.1 Serial system

The next model describes a *serial* system, where an item is processed by one server, followed by another server. The generator and the servers are decoupled by buffers. The model is depicted in Figure 10.3.

Figure 10.3: A generator, two buffers, two servers, and an exit.



The model can be described by:

In [12]:

```
def SerialSystemModel(ta, ts, N):
    env = Environment()
    a = Channel(env)
    b = Channel(env)
    c = Channel(env)
    d = Channel(env)
    e = Channel(env)
    G = Generator(env, a, ta)
    B1 = Buffer(env, a, b)
    S1 = Server(env, b, c, ts)
    B2 = Buffer(env, c, d)
    S2 = Server(env, d, e, ts)
    E = Exit(env, e, N)
    env.run(until=E)
```

executed in 6ms, finished 14:25:51 2021-11-08

The various processes are equal to those described in the example of the stochastic system model.

In [13]:

```
# Run the model
SerialSystemModel(3, 1, 10)
```

executed in 13ms, finished 14:25:51 2021-11-08

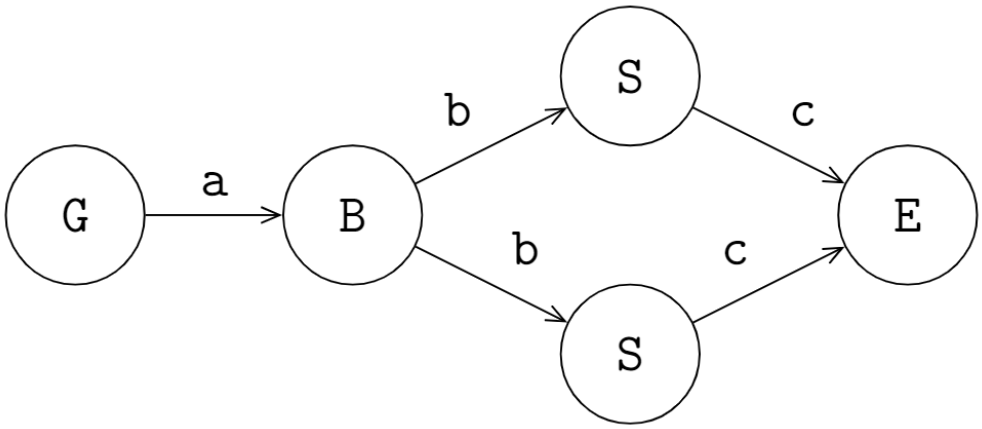
```
The Exit process received an item at t = 2.4 with flow time 2.4
The Exit process received an item at t = 8.4 with flow time 0.5
The Exit process received an item at t = 10.3 with flow time 2.3
The Exit process received an item at t = 15.8 with flow time 7.6
The Exit process received an item at t = 18.5 with flow time 7.2
The Exit process received an item at t = 18.8 with flow time 4.6
The Exit process received an item at t = 19.6 with flow time 5.1
The Exit process received an item at t = 20.0 with flow time 3.7
The Exit process received an item at t = 20.6 with flow time 1.8
The Exit process received an item at t = 26.8 with flow time 0.3
```

10.2.2 Parallel system

In a parallel system the servers are operating in parallel. Having several servers in parallel is useful for enlarging the processing capacity of the task being done, or for reducing the effect of break downs of servers (when a server breaks down, the other server continues with the task for other items). Figure 10.4 depicts the system.

Figure 10.4: A model with two parallel servers.

Figure 10.4: A model with two parallel servers.



Generator process `G` sends items via `a` to buffer process `B`, and process `B` sends the items in a first-in first-out manner to the servers `S1` and `S2`. Both servers send the processed items to the exit process `E` via channel `c`. The inter-arrival time and the two process times are assumed to be stochastic, and exponentially distributed. Items can pass each other, due to differences in processing time between the two servers.

If a server is free, and the buffer is not empty, an item is sent to a server. If both servers are free, one server will get the item, but which one cannot be determined beforehand. (How long a server has been idle is not taken into account.) The model is described by:

In [14]:

```
def ParallelSystemModel(ta, ts, N):
    env = Environment()
    a = Channel(env)
    b = Channel(env)
    c = Channel(env)
    G = Generator(env, a, ta)
    B = Buffer(env, a, b)
    S1 = Server(env, b, c, ts)
    S2 = Server(env, b, c, ts)
    E = Exit(env, c, N)
    env.run(until=E)
```

executed in 5ms, finished 14:25:51 2021-11-08

In [15]:

```
# Run the model
ParallelSystemModel(3, 1, 10)
```

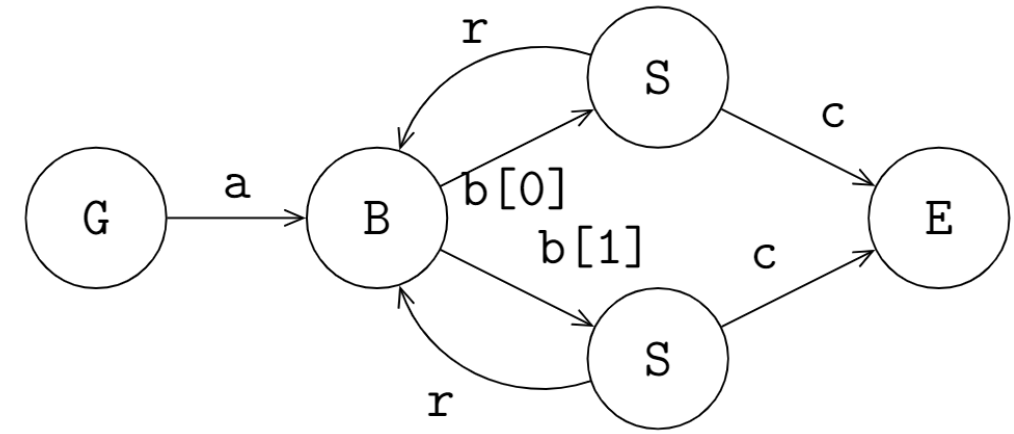
executed in 7ms, finished 14:25:51 2021-11-08

```
The Exit process received an item at t = 3.5 with flow time 3.5
The Exit process received an item at t = 5.6 with flow time 0.8
The Exit process received an item at t = 7.7 with flow time 0.6
The Exit process received an item at t = 9.6 with flow time 2.9
The Exit process received an item at t = 11.4 with flow time 1.4
The Exit process received an item at t = 33.8 with flow time 0.3
The Exit process received an item at t = 34.5 with flow time 0.2
The Exit process received an item at t = 35.0 with flow time 0.3
The Exit process received an item at t = 35.7 with flow time 0.4
The Exit process received an item at t = 36.4 with flow time 0.7
```

10.2.3 Parallel system with requests

To control which server gets the next item, each server must have its own channel from the buffer. In addition, the buffer has to know when the server can receive a new item. The latter is done with a 'request' channel, denoting that a server is free and needs a new item. The server sends its own identity as request, the requests are administrated in the buffer. The model is depicted in Figure 10.5.

Figure 10.5: A model with two parallel requesting servers.



In this model, the servers 'pull' an item through the line. The model is shown below. In this model, *list comprehension* is used for the initialization and running of the two servers. Via channel `r` an integer value, `0` or `1`, is sent to the buffer.

In [16]:

```
def RequestingParallelSystemModel(ta, ts, N):
    env = Environment()
    a = Channel(env)
    b = [Channel(env)] * 2
    c = Channel(env)
    r = Channel(env)
    G = Generator(env, a, ta)
    B = BufferRequesting(env, a, b, r)
    Ss = [ServerRequesting(env, b[j], c, r, ts, j) for j in range(2)]
    E = Exit(env, c, N)
    env.run(until=E)
```

executed in 5ms, finished 14:25:51 2021-11-08

The items received from generator `G` are stored in the buffer in list `xs`, the requests received from the servers are stored in list `ys`. The items and requests are removed form their respective lists in a first-in first-out manner. Process `B` is defined below.

If, there is an item present, *and* there is a server demanding for an item, the process sends the first item to the longest waiting server. The longest waiting server is denoted by variable `ys[0]`. The head of the item list is denoted by `xs[0]`. Assume the value of `ys[0]` equals `1`, then the expression `c_out[ys[0]].send(xs[0])` equals `c_out[1].send(xs[0])`, indicates that the first item of list `xs`, equals `xs[0]`, is sent to server `1`.

In [17]:

```
@process
def BufferRequesting(env, c_in, c_out, c_r):
    xs = []
    ys = []
    while True:
        sending = c_out[ys[0]].send(xs[0]) if (len(xs)>0 and len(ys)>0) else None
        receiving = c_in.receive()
        request = c_r.receive()
        z = yield env.select(receiving, request, sending)
        if selected(receiving):
            xs = xs + [z]
        if selected(request):
            ys = ys + [z]
        if selected(sending):
            xs = xs[1:]
            ys = ys[1:]
```

executed in 5ms, finished 14:25:51 2021-11-08

The server first sends a request via channel `r` to the buffer, and waits for an item. The item is processed, and sent to exit process `E`.

In [18]:

```
@process
def ServerRequesting(env, c_in, c_out, c_r, ts, k):
    u = lambda: random.exponential(ts)
    while True:
        yield env.execute(c_r.send(k))
        x = yield env.execute(c_in.receive())
        delay = u()
        yield env.timeout(delay)
        yield env.execute(c_out.send(x))
```

executed in 14ms, finished 14:25:51 2021-11-08

Again the exit process `E` remains unchanged. So we can now run the model to see the results.

In [19]:

```
# Run the model
RequestingParallelSystemModel(3, 1, 10)
```

executed in 12ms, finished 14:25:51 2021-11-08

```
The Exit process received an item at t = 0.4 with flow time 0.4
The Exit process received an item at t = 3.5 with flow time 2.0
The Exit process received an item at t = 6.0 with flow time 1.1
The Exit process received an item at t = 7.7 with flow time 1.6
The Exit process received an item at t = 8.4 with flow time 2.5
The Exit process received an item at t = 10.1 with flow time 0.5
The Exit process received an item at t = 10.3 with flow time 0.0
The Exit process received an item at t = 12.8 with flow time 0.0
The Exit process received an item at t = 14.8 with flow time 0.1
The Exit process received an item at t = 16.1 with flow time 1.3
```

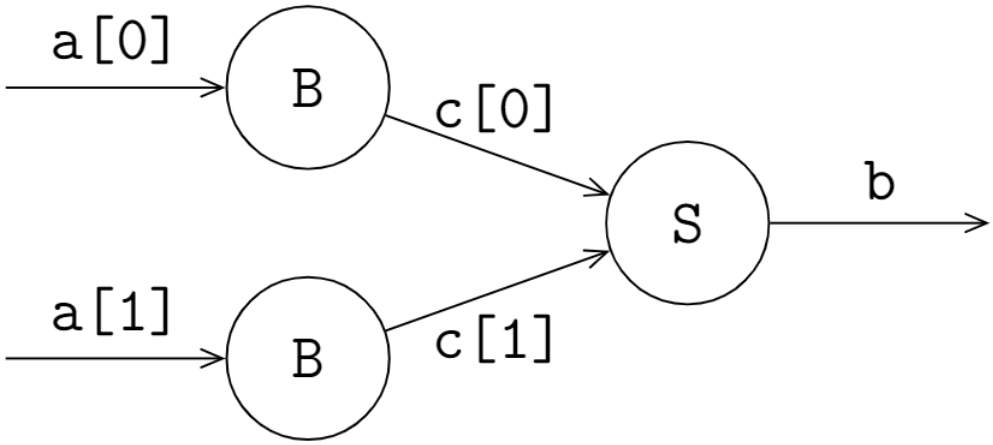
10.3 Assembly

In assembly systems, components are assembled into bigger components. These bigger components are assembled into even bigger components. In this way, products are built, e.g. tables, chairs, computers, or cars. In this section some simple assembly processes are described. These systems illustrate how assembling can be performed: in industry these assembly processes are often more complicated.

An assembly work station for two components is shown in Figure 10.6.

Figure 10.6: Assembly for two components.

Figure 10.6: Assembly for two components.



We first create the model for a system with assembly. Both buffers are preceded by a generator. The server is succeeded by a single exit. The model is as follows:

In [20]:

```
def Assemble2PartsSystemModel(ta, ts, N):
    env = Environment()
    a = [Channel(env)] * 2
    c = [Channel(env)] * 2
    b = Channel(env)
    Gs = [Generator(env, a[j], ta) for j in range(2)]
    Bs = [Buffer(env, a[j], c[j]) for j in range(2)]
    S = ServerAssembly(env, c, b)
    E = Exit(env, b, N)
    env.run(until=E)
```

executed in 5ms, finished 14:25:51 2021-11-08

The assembly process server `S` is preceded by buffers. The server receives an item from each buffer `B`, before starting assembly. The received items are assembled into one new item, a list of its (sub-)items. The description of the assembly server is shown below.

The process takes a list of channels `c` (which corresponds to `c_in` in the process definition below) to receive items from the preceding buffers. The output channel `b` (which corresponds to `c_out` in the process definition below) is used to send the assembled component away to the next process.

First, the assembly process receives an item from both buffers. All buffers are queried at the same time, since it is unknown which buffer has components available. If the first buffer reacts first, and sends an item, it is received with channel `c[0]` and stored in `v[0]` in the first alternative. The next step is then to receive the second component from the second buffer, and store it (`v[1] = yield env.execute(c_in[1].receive())`). The second alternative does the same, but with the channels and stored items swapped. When both components have been received, the assembled product is sent away.

In [21]:

```
@process
def ServerAssembly(env, c_in, c_out):
    v = [None, None]
    while True:
        receive_part1 = c_in[0].receive()
        receive_part2 = c_in[1].receive()
        x = yield env.select(receive_part1, receive_part2)

        if selected(receive_part1):
            v[0] = x
            v[1] = yield env.execute(c_in[1].receive())

        if selected(receive_part2):
            v[1] = x
            v[0] = yield env.execute(c_in[0].receive())

        yield env.execute(c_out.send(v))
```

executed in 7ms, finished 14:25:51 2021-11-08

We make a slight alteration to the exit model to show that the parts have indeed been assembled.

In [22]:

```
@process
def Exit(env, c_in, N):
    for i in range(N):
        x = yield env.execute(c_in.receive())
        print(f"The Exit process received an item at t = {env.now:.1f} consisting o
```

executed in 11ms, finished 14:25:51 2021-11-08

Now we can run this model with `ta=3`, `ts=1`, and `N=10`.

In [23]:

```
# Run the model
Assemble2PartsSystemModel(3, 1, 10)
```

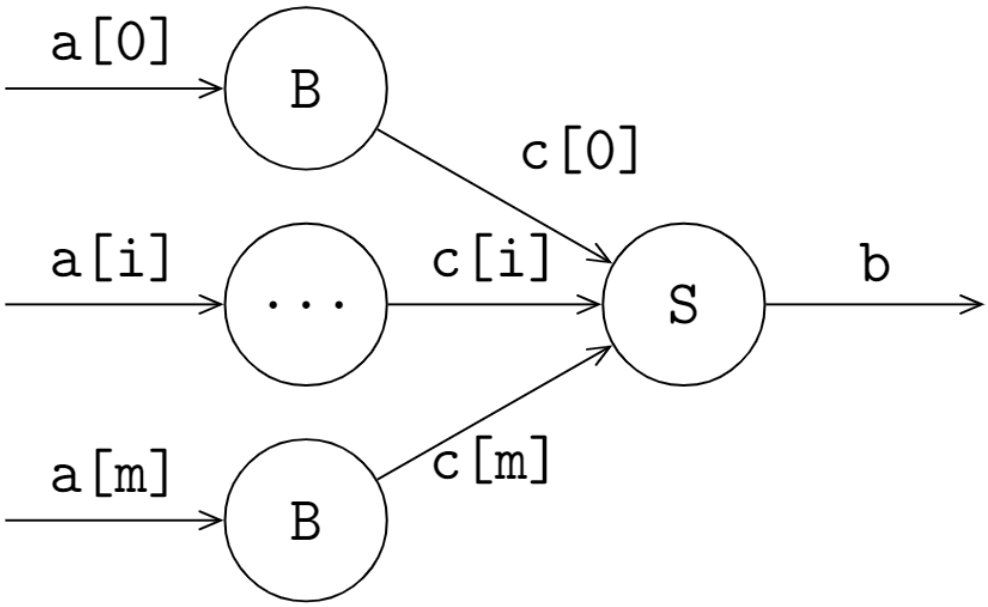
executed in 15ms, finished 14:25:51 2021-11-08

```
The Exit process received an item at t = 0.0 consisting of 2 parts.
The Exit process received an item at t = 2.4 consisting of 2 parts.
The Exit process received an item at t = 5.2 consisting of 2 parts.
The Exit process received an item at t = 8.1 consisting of 2 parts.
The Exit process received an item at t = 8.4 consisting of 2 parts.
The Exit process received an item at t = 11.9 consisting of 2 parts.
The Exit process received an item at t = 12.3 consisting of 2 parts.
The Exit process received an item at t = 14.1 consisting of 2 parts.
The Exit process received an item at t = 16.7 consisting of 2 parts.
The Exit process received an item at t = 19.8 consisting of 2 parts.
```

10.3.1

A generalized assembly work station for `n` components is depicted in Figure 10.2.

Figure 10.7: Assembly for `n` components, with `m=n-1`.



Again, we first create the systen model. We substitute the buffers and servers with workstation submodel `W`, which is preceded by `n` generators. The workstation is succeeded by a single exit. The model is as follows:

In [24]:

```
def AssembleMPartsSystemModel(ta, ts, N, n):
    env = Environment()
    a = [Channel(env)] * n
    c = [Channel(env)] * n
    b = Channel(env)
    Gs = [Generator(env, a[j], ta) for j in range(n)]
    W = WorkstationAssembly(env, a, b)
    E = Exit(env, b, N)
    env.run(until=E)
```

executed in 7ms, finished 14:25:51 2021-11-08

The entire work station submodel (the combined buffer processes and the assembly server process) is shown below.

The size of the list of channels `c_in` is determined during initialization of the workstation. This size is used for the generation of the process buffers, and the accompanying channels.

In [25]:

```
def WorkstationAssembly(env, c_in, c_out):
    n = len(c_in)
    c_toS = [Channel(env)] * n
    Bs = [Buffer(env, c_in[i], c_toS[i]) for i in range(n)]
    Ss = ServerAssemblyMParts(env, c_toS, c_out)
```

executed in 5ms, finished 14:25:51 2021-11-08

The assembly server process works in the same way as before, except for a generic `n` components, it is impossible to write a select statement explicitly. Instead, the list of communication events `receive_parts` is defined using *list comprehension*, which combined with `env.select(*receive_parts)` is used to unfold the alternatives.

The received components are again in `v`. Item `v[i]` is received from channel `c[i]`. The indices of the channels that have not provided an item are in the list `rec`. Initially, it contains all channels `0 ... n`, that is, `rec = list(range(n))`. While `rec` still has a channel index to monitor, the `[c_in[i].receive() if (i in rec) else None for i in range(n)]` contains all possible communication events that are required to finalize the assembly process. For example, if `rec` contains `[0, 1, 4]` for `n=5`, then `receive_parts =[c_in[i].receive() if (i in rec) else None for i in range(n)]` is equivalent to:

```
receive_parts = [c_in[0].receive(), c_in[1].receive(), None, None, c_in[4].re
ceive()]
```

.

Which means that `yield env.select(*receive_parts)` will try to receive an item over either channels `c_in[0]`, `c_in[1]` or `c_in[4]`.

After receiving an item, the index of the channel is removed from `rec` to prevent receiving a second item from the same channel. When all items have been received, the assembled component is sent away with `yield env.execute(c_out.send(v))`.

In [26]:

```
@process
def ServerAssemblyMParts(env, c_in, c_out):
    n = len(c_in)
    v = [None]*n
    while True:
        rec = list(range(n))
        while len(rec)>0:
            receive_parts = [c_in[i].receive() if (i in rec) else None for i in ran
            x = yield env.select(*receive_parts)
            for j in range(n):
                if selected(receive_parts[j]):
                    v[j] = x
                    rec.remove(j)
        yield env.execute(c_out.send(v))
```

executed in 11ms, finished 14:25:51 2021-11-08

The exit model remains the same as in the previous assembly system model.

In practical situations these assembly processes are performed in a more cascading manner: two or three components are 'glued' together in one assemble process, followed in the next process by another assembly process.

In [27]:

```
# Run the model
AssembleMPartsSystemModel(3, 1, 10, 4)
```

executed in 19ms, finished 14:25:51 2021-11-08

```
The Exit process received an item at t = 0.0 consisting of 4 parts.
The Exit process received an item at t = 3.3 consisting of 4 parts.
The Exit process received an item at t = 5.0 consisting of 4 parts.
The Exit process received an item at t = 7.2 consisting of 4 parts.
The Exit process received an item at t = 9.3 consisting of 4 parts.
The Exit process received an item at t = 12.9 consisting of 4 parts.
The Exit process received an item at t = 19.6 consisting of 4 parts.
The Exit process received an item at t = 25.8 consisting of 4 parts.
The Exit process received an item at t = 27.8 consisting of 4 parts.
The Exit process received an item at t = 29.1 consisting of 4 parts.
```

10.4 Exercises

10.4.1

Predict the resulting throughput and flow time for a deterministic case like in Section 10.1.1, with `ta = 4` and `ts = 5`. Verify the prediction with an experiment, and explain the result.

10.4.2

Extend the model of Exercise 1 in Section 9.5 with a single deterministic server taking `4.0` time units to model the production capacity of the factory. Increase the number of products inserted by the generator, and measure the average flow time for

1. A FIFO buffer with control policy `low = 0` and `high = 1`.
2. A FIFO buffer with control policy `low = 1` and `high = 4`.
3. A LIFO buffer with control policy `low = 1` and `high = 4`.

In [ ]:

```
@dataclass
class Product:
    id: int
    entrytime: float

@process
def Generator(env, c_out, c_signal, N):
    for i in range(N):
        yield env.execute(c_signal.receive())
        x = Product(id = i, entrytime = env.now)
        yield env.execute(c_out.send(x))

@process
def Exit(env, c_in, c_signal):
    mean_flowtime = 0.0
    i = 1
    while True:
        yield env.execute(c_signal.receive())
        x = yield env.execute(c_in.receive())
        flowtime = env.now - x.entrytime
        mean_flowtime = (i - 1) / i * mean_flowtime  +  flowtime / i
        print(f"The Exit process received {x.id}, with flowtime {flowtime:.2f}. The
        i = i+1

@process
def Controller(env, c_signal_gen, c_signal_exit, low, high):
    count = 0
    while True:
        while count < high:
            yield env.execute(c_signal_gen.send())
            count = count+1
        while count > low:
            yield env.execute(c_signal_exit.send())
            count = count-1

def model(low, high, N):
    env = Environment()
    sg = Channel(env)
    se = Channel(env)
    gf = Channel(env)
    fe = Channel(env)
    G = Generator(env, gf, sg, N)
    F = Factory(env, gf, fe)
    E = Exit(env, fe, se)
    C = Controller(env, sg, se, low, high)
    env.run()
```

A.

In [ ]:

```
# The factory is a submodel which contains both a FIFO buffer and a server
def Factory(env, c_in, c_out):
    ...
```

In [ ]:

```
model(low=0, high=1, N=..)
```

B.

In [ ]:

```
# 2b
model(low=1, high=4, N=..)
```

C.

In [ ]:

```
# 2c: The factory is a submodel which contains both a LIFO buffer and a server
def Factory(env, c_in, c_out):
    ...
```

In [ ]:

```
# 2c
model(low=1, high=4, N=..)
```

10.5 Answers to exercises

Answer to 10.4.1

[Click for the answer to 10.4.1]

Answer to 10.4.2

[Click for the answer to 10.4.2]