Unit 3: Reusing code – Functions, modules and packages

Richard Foltyn
University of Glasgow

September 10, 2022

Contents

1 Reusing code: Functions, modules and packages

In this unit, we learn how to build reusable code with functions. We will also briefly discuss modules and packages.

1.1 Functions

Functions are used to implement code that performs a narrowly defined task. We use functions for two reasons:

1. A function can be called repeatedly without having to write code again and again.
2. Even if a function is not called frequently, functions allow us to write code that is "shielded" from other code we write and is called via a clean interface. This helps to write more robust and error-free code.

Functions are defined using the `def` keyword, and the function body needs to be an indented block:

[1]:
```
def func():
    print('func called')

# invoke func without arguments
func()
```

```
func called
```

1.1.1 Arguments

Functions can have an arbitrary number of positional arguments (also called parameters).

[2]:
```
# Define func to accept argument x
def func(x):
    print(f'func called with argument {x}')

# call function with various arguments.
```

```
func(1)
func('foo')
```

```
func called with argument 1
func called with argument foo
```

1.1.2 Return values

Functions can also return values to their caller using the `return` statement.

[3]:
```
def func(x):
    return x * 2.0

result = func(1.0)
print(result)        # prints 2.0
```

```
2.0
```

A `return` statement without any argument immediately exits the functions. The default return value is the special type `None`.

A function can return multiple values which are then automatically collected into a tuple:

[4]:
```
def func():
    return 1, 2, 3

values = func()          # call func(), get tuple of values
type(values)
```

[4]:
```
tuple
```

Python supports "unpacking" of tuples, lists, etc. We can use this to directly assign names to multiple return values:

[5]:
```
def func():
    return 'a', 'b', 'c'

value1, value2, value3 = func()      # call func(), unpack return values
print(f'Value 1: {value1}, Value 2: {value2}, Value 3: {value3}')
```

```
Value 1: a, Value 2: b, Value 3: c
```

1.1.3 Accessing data from the outer scope

A function need not have arguments or a return value, but that limits its usefulness somewhat. However, a function can access outside data which is defined in the so-called outer scope:

[6]:
```
x = 1.0
def func():
    # Read x from outer scope
    print(f'func accessing x from outer scope: x = {x}')

# prints value of x from within func()
func()
```

```
func accessing x from outer scope: x = 1.0
```

We can write functions without any arguments that only operate on outside data. However, this is terrible programming practice and should be avoided in most cases.

Because functions can operate on external data, they are not analogous to mathematical functions. If we write $f(x)$, we usually mean that f is a function of x only (and possibly some constant parameters). By definition we must have

$$x_1 = x_2 \implies f(x_1) = f(x_2),$$

i.e., a function always returns the same value when called with the same parameters. However, this is not the case in Python or most other programming languages:

[7]:
```
a = 1.0
def func(x):
    return a*x

x = 1.0
print(func(x))       # prints 1.0

a = 2.0
print(func(x))       # x unchanged, but prints 2.0
```

```
1.0
2.0
```

1.1.4 More on arguments

Default arguments

Python offers an extremely convenient way to specify default values for arguments, so these need not be passed when the function is called:

[8]:
```
# define function with a default value for argument alpha
def func(x, alpha=1.0):
    return x * alpha

print(func(2.0))         # uses default value for alpha
print(func(2.0, 1.0))    # explicitly specified optional argument
print(func(2.0, 3.0))    # use some other value for alpha
```

```
2.0
2.0
6.0
```

Keyword (or named) arguments

Arguments don't need to be provided in the same order as specified in the function signature. We can use argument names to explicitly assign values to the corresponding argument.

[9]:
```
def func(arg1, arg2):
    print(f'arg1: {arg1}, arg2: {arg2}')

func(1, 2)               # Call using purely positional arguments
func(arg1=1, arg2=2)     # Use argument names to explicitly assign values
func(arg2=2, arg1=1)     # With keyword arguments, the order does not matter!
```

```
arg1: 1, arg2: 2
arg1: 1, arg2: 2
arg1: 1, arg2: 2
```

Arbitrary number of optional arguments

Python supports functions which accept an arbitrary number of positional and keyword arguments. This is accomplished via two special tokens:

- `*args`: collects any number of "excess" *positional arguments* and packs them into a tuple.
- `**kwargs`: collects any number of "excess" *keyword arguments* and packs them into a dictionary. Needs to be placed at the end of the argument list!

Examples:

[10]:
```
# Define function with mandatory, optional, optional positional
# and optional keyword arguments

def func(x, opt='default', *args, **kwargs):
    print(f'Required argument x: {x}')
    print(f'Optional argument opt: {opt}')
    if args:
        # if the tuple 'args' is non-empty, print its contents
        print('Optional positional arguments:')
        for arg in args:
            print(f'  {arg}')
    if kwargs:
        # if the dictionary 'kwargs' is non-empty, print its contents
        print('Optional keyword arguments:')
        for key, value in kwargs.items():
            print(f'  {key}: {value}')
```

[11]:
```
# Call only with required argument
func(0)
```

```
Required argument x: 0
Optional argument opt: default
```

[12]:
```
# Call with required and optional arguments
func(0, 'optional')
```

```
Required argument x: 0
Optional argument opt: optional
```

[13]:
```
# Call with required and optional arguments, and
# optional positional arguments
func(0, 'optional', 1, 2, 3)
```

```
Required argument x: 0
Optional argument opt: optional
Optional positional arguments:
  1
  2
  3
```

[14]:
```
# Call with required and optional arguments, and
# optional positional and keyword arguments
func(0, 'optional', 1, 2, 3, arg1='value1', arg2='value2')
```

```
Required argument x: 0
Optional argument opt: optional
Optional positional arguments:
  1
  2
  3
Optional keyword arguments:
  arg1: value1
  arg2: value2
```

We don't even need to specify arguments in the order they are defined in the function, except for optional positional arguments, since these have no argument names. We can just use the `name=value` syntax:

[15]:
```
# call func() with interchanged argument order
func(opt='optional value', x=1)
```

```
Required argument x: 1
Optional argument opt: optional value
```

Note, however, that in a function call any positional arguments must come first and those passed as `name=value` pairs last:

[16]:
```
x = 1

# this will not work, cannot specify positional arguments last
func(opt='optional value', x)
```

```
  Input In [16]
    func(opt='optional value', x)
                               ^
SyntaxError: positional argument follows keyword argument
```

The same applies for optional arguments passed in via `*args` and `**kwargs`:

[17]:
```
# fails because arguments collected in *args must
# be specified before arguments collected in **kwargs!
func(1.0, 'opt', arg1='value1', arg2='value2', 1, 2, 3)
```

```
  Input In [17]
    func(1.0, 'opt', arg1='value1', arg2='value2', 1, 2, 3)
                                                         ^
SyntaxError: positional argument follows keyword argument
```

1.1.5 Modifying data in the outer scope

So far, we covered read-only access to data defined outside of a function and relied on return values to pass results back to the caller. However, it is possible to directly *modify* data in the outer scope, even though this should usually be avoided:

- Using arguments and return values clearly defines a function's interface, there are no unpleasant surprises.
- Conversely, if a function starts modifying values in the caller's environment, there is no way to be sure what the function is changing in the outer scope
  other than examining its source code.

Consider first the following attempt to modify a value defined outside of the function:

[18]:
```
var = 'outer scope'

# Create function, assign value to var
def modify_var():
    var = 'inner scope'

print(var)
modify_var()
print(var)
```

```
outer scope
outer scope
```

This code prints `'outer scope'` twice. What happened? Without any further instructions, the assignment inside the function creates a *local* variable `var` that is completely disconnected from `var` in the outer scope!

We need to use the `global` statement to tell Python to instead assign to a variable in the global (outer) scope:

[19]:
```
var  = 'outer scope'

def modify_var():
    global var
    var = 'inner scope'

print(var)
modify_var()
print(var)
```

```
outer scope
inner scope
```

The second output now reads `'inner scope'`.

Note that `global` in Python actually means global to a module, i.e., a symbol that is defined at the top level within a module (we discuss modules below). There are no truly global variables in Python unlike in languages such as C.

The requirement that the name in the `global` statement refers to a global variable has subtle implications. Consider the following example of two *nested* functions:

[20]:
```
def outer():
    var = 'outer function'

    def inner():
        # Bind var to global name var
        global var
        var = 'inner function'

    print(var)
    inner()
    print(var)

outer()
```

```
outer function
outer function
```

Surprisingly, the code above prints `'outer function'` twice. The reason is that `var` defined in `outer()` is *not* a global variable as it was not defined at the top level within a module. Instead, it is a *local* variable in `outer()`.

For such scenarios, Python has the `nonlocal` statement which works similarly to `global` except that it operates on a name in the immediate outer scope, irrespective of whether this outer scope is another function or the module itself.

We can use `nonlocal` to get the desired behaviour:

[21]:
```
def outer():
    # var is in outer's local scope
    var = 'outer function'

    def inner():
        # bind var to name in immediate outer scope,
        # which is the local scope of outer()
        nonlocal var
        var = 'inner function'

    print(var)
    inner()
    print(var)

outer()
```

```
outer function
inner function
```

1.1.6 Pass by value or pass by reference?

Can functions modify their arguments? This questions usually comes down to whether a function call uses *pass by value* or *pass by reference*:

- *pass by value* means that a copy of every argument is created before it is passed into the function. A function therefore cannot modify a value in the caller's environment.
- *pass by reference* means that only a reference to a value is passed to the function, so the function can directly modify values at the call site.

This programming model is used in languages such as C (pass by value) or Fortran (pass by reference), but not in Python. Sloppily speaking, we can say that in Python a reference ("variable name") is passed by value. This means assigning a different value to an argument ("the reference") within a function has no effect outside of the function:

[22]:
```
def func(x):
    # x now points to something else
    x = 1.0

x = 123
func(x)

x       # prints 123, x in the outer scope is unchanged
```

[22]:
```
123
```

However, if a variable is a mutable object (such as a `list` or a `dict`), the function can use its own copy of the reference to that object to modify the object even in the outer scope:

[23]:
```
def func(x):
    # uses reference x to modify list object outside of func()
    x.append(4)

lst = [1,2,3]
func(lst)
lst     # prints [1,2,3,4]
```

[23]:
```
[1, 2, 3, 4]
```

Nevertheless, even for mutable objects the rule from above applies: when a new value is *assigned* to a named argument, that name then references a different object, leaving the original object unmodified:

[24]:
```
def func(x):
    # this does not modify object in outer scope,
    # x now references a new (local) object.
    x = ['a', 'b', 'c']

lst = [1,2,3]
# pass list, which is mutable and can thus be changed in func()
func(x)

lst     # prints [1,2,3]
```

[24]:
```
[1, 2, 3]
```

For immutable objects such as tuples, the reference passed to the function of course cannot be used to modify the object inside the function:

[25]:
```
def func(x):
    x[0] = 'modified in func'

items = (1, 2, 3)           # create tuple of integers
func(items)
```

```
TypeError: 'tuple' object does not support item assignment
```

Passing in a mutable collection such as a list, however, works as expected:

[26]:
```
items = [1, 2, 3]

func(items)

items
```

[26]:
```
['modified in func', 2, 3]
```

1.1.7 Methods

Methods are simply functions that perform an action on a particular object which they are bound to. We will not write methods in this course ourselves (they are part of what's called object-oriented programming), but we frequently use them when we invoke actions on objects such as lists:

[27]:
```
# Create a list
lst = [1,2,3]

# append() is a method of the list class and can be invoked
# on list objects.
lst.append(4)
lst
```

[27]:
```
[1, 2, 3, 4]
```

1.1.8 Functions as objects

Functions are objects in their own right, which means that you can perform various operations with them:

- Assign a function to a variable.
- Store functions in collections.
- Pass function as an argument to other functions.

Examples:

[28]:
```
def func1(x):
    print(f'func1 called with argument {x}')

def func2(x):
    print(f'func2 called with argument {x}')


# List of functions
funcs = [func1, func2]

# Assign functions to variable f
for f in funcs:
    # call function referenced by f
    f('foo')
```

```
func1 called with argument foo
func2 called with argument foo
```

[29]:
```
# Pass one function as argument to another function
func1(func2)
```

```
func1 called with argument <function func2 at 0x7fea14255b40>
```

1.1.9 lambda expressions

You can think of lambda expressions as light-weight functions. The syntax is

`lambda x: <do something with x>`

The return value of a lambda expression is whatever its body evaluates to. There is no need (or possibility) to explicitly add a `return` statement.

One big difference to regular functions is that lambda expressions are *expressions*, not statements.

- At this point we gain little from a technical discussion on *statements* vs *expressions*. Loosely speaking, statements are one level above expressions in the Python syntax hierarchy, and the language puts restrictions on where statements can appear. Function definitions, `for` and `while` loops, and `if/elif/else` blocks are statements, among others.
- Conversely, *expressions* are more flexible and can appear basically anywhere. They usually evaluate to some object that can be assigned, passed to a function, etc., whereas statements usually can't.

The take-away is that we can fiddle in lambda expressions almost anywhere, even as arguments in function calls!

For example, we might have a function that applies some algebraic operation to its arguments, and the operation can be flexibly defined by the caller.

[30]:
```
def func(items, operation=lambda z: z + 1):
    # default operation: increment value by 1
    result = [operation(i) for i in items]
    return result

numbers = [1.0, 2.0, 3.0]
# call with default operation
func(numbers)                # prints [2.0, 3.0, 4.0]
```

[30]:
```
[2.0, 3.0, 4.0]
```

[31]:
```
# We can also use lambda expressions to specify
# an alternative operation directly in the call!

func(numbers, lambda x: x**2.0)      # prints [1.0, 4.0, 9.0]
```

[31]:
```
[1.0, 4.0, 9.0]
```

While we could have defined the operation using a "regular" function statement, this is shorter.

1.2 Modules and packages

1.2.1 Modules

Modules allow us to further encapsulate code that implements some particular functionality.

- Each Python file (with the extension `.py`) automatically corresponds to a module of the same name.
- Objects defined within such a module are by default not visible outside of the module, thus helping to avoid naming conflicts.

To actually demonstrate the usage of modules, we need to use files outside of this notebook. To this end, there is an additional Python file in the current directory:

```
lectures/
    unit03_mod.py
```

The module `unit03_mod.py` contains the following definitions:

```
# Contents of unit03_mod.py

# global variable in this module
var = 'Variable defined in unit03_mod'

# global function in this module
def func():
    print(f'func in module unit03_mod called')
```

Module search path

Before getting into the details, we first need to verify that we can import the module `unit03_mod` using the `import` statement:

[32]:
```
import unit03_mod
```

Depending on where exactly you are running this code, the above import statement might fail with a `ModuleNotFoundError` (if no error was raised you can skip the rest of this section). This happens whenever the directory in which the module resides is not in the *module search path* used by Python.

To fix this, check the module search path as follows:

[33]:
```
import sys
sys.path
```

[33]:
```
['/home/richard/repos/teaching/MLFP-ECON5130/lectures',
 '/home/richard/.conda/envs/py3-default/lib/python310.zip',
 '/home/richard/.conda/envs/py3-default/lib/python3.10',
 '/home/richard/.conda/envs/py3-default/lib/python3.10/lib-dynload',
 '',
 '/home/richard/.conda/envs/py3-default/lib/python3.10/site-packages',
 '/home/richard/.conda/envs/py3-default/lib/python3.10/site-
packages/PyQt5_sip-12.11.0-py3.10-linux-x86_64.egg']
```

If the `lectures/` directory is not included in this list, you can add it manually. For example, this notebook is executed in the git repository's root directory, you need to exectute

[34]:
```
import sys
# add lectures/ directory using a relative path
sys.path.append('./lectures/')
```

Importing symbols

We now want to use `func` and `var` in our notebook. However, by default these symbols are not visible and first need to be imported. We can do this in several ways:

1. We can import the module and use fully qualified names to reference objects from `unit03_mod`.
2. We can select which names from `unit03_mod` should be directly accessible.

The first variant looks like this:

[35]:
```
import unit03_mod

# Access variable defined in unit03_mod
print(unit03_mod.var)

# Call function defined in unit03_mod
```

```
unit03_mod.func()
```

```
Variable defined in unit03_mod
func in module unit03_mod called
```

If a symbol from `unit03_mod` is used frequently, we might want to make it accessible without the `unit03_mod` prefix. This is the second variant:

[36]:
```
from unit03_mod import var, func

# Access variable defined in unit03_mod
print(var)

# Call function defined in unit03_mod
func()
```

```
Variable defined in unit03_mod
func in module unit03_mod called
```

What if our notebook itself defines a function `func()` which would overwrite the reference to the one imported from `unit03_mod`, as in the following example?

[37]:
```
from unit03_mod import func

# Calls func() defined in unit03_mod
func()

# overwrites definition from unit03_mod with local version
def func():
    print('func in notebook called')

# Calls func() defined in notebook
func()
```

```
func in module unit03_mod called
func in notebook called
```

In such a scenario, we can assign aliases to imported symbols using `as`:

[38]:
```
from unit03_mod import func as imported_func      # The function formerly known
                                                  # as func is now imported_func

def func():
    print('func in notebook called')

# call our own func
func()

# call func from module unit03_mod
imported_func()
```

```
func in notebook called
func in module unit03_mod called
```

We can even alias the module name itself, as we frequently do with widely used modules such as `numpy`:

[39]:
```
import unit03_mod as u4m

u4m.func()    # call function from module unit03_mod
```

```
func in module unit03_mod called
```

[40]:
```
# universal convention to import numpy like this
import numpy as np
```

1.2.2 Packages

Packages are roughly speaking collections of modules and a little magic on top. We will not be creating packages, but we have already been using them: basically everything besides the built-in functions is defined in some package. For example, the NumPy library is a collection of packages.

1.3 Exercises

1.3.1 Exercise 1: Sign function

Implement a function `sign` which returns the following values:

$$sign(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Test your function on a negative, zero and positive argument.

1.3.2 Exercise 2: Sum of arbitrary number of elements

Create a function called `my_sum` which accepts an arbitrary number of arguments (possibly zero) and returns their sum. Assume that all arguments are numeric.

Test your function with the following arguments:

```
my_sum(10.0)     # one argument
my_sum(1,2,3)    # multiple arguments
my_sum()         # no arguments
```

Make sure that in the last case your function returns zero, which is the sum over an empty set.

1.3.3 Exercise 3: Fibonacci sequence

A classical introductory exercise to programming is to write a function that returns the first n terms of the Fibonacci sequence. The i-th element of this sequence is the integer x_i defined as

$$x_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ x_{i-1} + x_{i-2} & \text{else} \end{cases}$$

Write a function `fibonacci(i)`,

```
def fibonacci(i):
    ...
```

which returns the i-th item in the sequence using recursion. A recursive function is a function that calls itself to perform (part of) its task, i.e., you should compute x_i like this:

```
xi = fibonacci(i-1) + fibonacci(i-2)
```

Use this function to compute the first 10 elements of this sequence with a list comprehension.

1.3.4 Exercise 4: Factorials

1. Implement a function that computes the factorial of a non-negative integer n defined as $n! = \prod_{i=1}^{n} i$. Keep in mind that this definition implies that $0! = 1$. Use the list comprehension syntax to create a tuple that contains the factorials for the integers $n = 1, \ldots, 10$.

   Hint: The factorial can be written as a recurrence relation $n! = n \cdot (n-1)!$, which you can use to implement the recursive function.

2. Provide an alternative implementation that does not rely on recursion, but instead uses NumPy's `prod()` function to compute the product of a sequence of numbers. Again, create a `tuple` that contains the factorials for the integers $n = 1, \ldots, 10$ using a list comprehension.

   Hint 1: To compute the product of the integers $i, i+1, \ldots, j$, you can use `np.prod(range(i, j+1))`.

   Hint 2: The product of an empty set is 1, which is what `np.prod()` returns.

1.3.5 Exercise 5: Bisection root-finding algorithm (advanced)

We revisit the binary search algorithm from unit 2, this time applied to finding the root of a continuous function. This is called the bisection method.

Implement a function `bisection(f, a, b, tol, xtol)` which finds the root of the function $f(x)$, i.e., the value x_0 where $f(x_0) = 0$, on the interval $[a, b]$. Assume that $a < b$ and that the function values $f(a)$ and $f(b)$ have opposite signs.

Test your implementation using the function $f(x) = x^2 - 4$ on the interval $[-3, 0]$, which has a (unique) root at $x_0 = -2$.

Hint: The bisection algorithm proceeds as follows:

1. Define tolerance levels $\epsilon > 0$ and $\epsilon_x > 0$. The algorithm completes successfully whenever we have either $|f(x_0)| < \epsilon$ or $|b - a| < \epsilon_x$.
2. Main loop of the algorithm:
   1. Compute the midpoint $x_m = (a + b)/2$
   2. Compute function value $f_m = f(x_m)$
   3. If either $|f_m| < \epsilon$ or $|b - a| < \epsilon_x$, accept x_m as the solution and exit.
   4. Otherwise, update either a or b:
      1. If $sign(f(b)) = sign(f_m)$, set $b = x_m$
         Hint: One way to check whether two non-zero values have the same sign is to check if $f(b) \cdot f_m > 0$.
      2. Otherwise, $a = x_m$
   5. Proceed to next iteration of main loop.

1.4 Solutions

1.4.1 Solution for exercise 1

[41]:
```
import numpy as np

def sign(x):
    if x < 0.0:
        return -1.0
    elif x == 0.0:
        return 0.0
    elif x > 0.0:
        return 1.0
    else:
        # Argument is not a proper numerical value, return NaN
        # (NaN = Not a Number)
        return np.nan

# Test on a few values
print(sign(-123))
print(sign(0))
print(sign(12345))
```

```
-1.0
0.0
1.0
```

Note that NumPy has a "proper" sign function, `np.sign()`, which implements the same logic but is more robust, accepts array arguments, etc.

1.4.2 Solution for exercise 2

For a function to accept an arbitrary number of elements, we need to declare an `*args` argument.

One possible implementation of `my_sum()` looks as follows:

[42]:
```
def my_sum(*args):
    # Initialise sum to 0
    s = 0
    for x in args:
        s += x
    return s


# Test with built-in range() object
print(my_sum(10.0))
print(my_sum(1, 2, 3))
print(my_sum())
```

```
10.0
6
0
```

Of course in real code we would use the built-in function `sum()`, or preferably the NumPy variant `np.sum()`:

[43]:
```
import numpy as np

print(np.sum(10.0))

# Need to pass argument as collection
print(np.sum((1, 2, 3)))

# np.sum() cannot be invoked without arguments, but we can
# call it with an empty tuple ()
np.sum(())
```

```
10.0
6
```

[43]:
```
0.0
```

1.4.3 Solution for exercise 3

The recursive definition of `fibonacci(i)` could look like this:

[44]:
```
def fibonacci(i):
    if i == 0:
        # No recursion needed
        xi = 0
    elif i == 1:
        # No recursion needed
        xi = 1
    else:
        # Assume that i > 1. We will learn later how to
        # return an error if this is not the case.
        # Use recursion to compute the two preceding values
        # of the sequence.
        xi = fibonacci(i-1) + fibonacci(i-2)
```

```
    return xi

# Compute the first 10 elements of the sequence using a list comprehension
first10 = [fibonacci(i) for i in range(10)]
first10
```

[44]:
```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Note that this is a terribly inefficient way to compute things, as the same elements of the sequence will needlessly be calculated over and over again.

Also, Python has a built-in recursion limit, so you cannot call a function recursively arbitrarily many times. You can find out what this limit is as follows:

[45]:
```
import sys
print(sys.getrecursionlimit())
```

```
3000
```

1.4.4 Solution for exercise 4

The following code shows a function computing the factorial $n!$ using recursion:

[46]:
```
def factorial(n):
    if n == 0:
        return 1
    else:
        # Use recursion to compute factorial
        return n * factorial(n-1)

fact10 = tuple(factorial(n) for n in range(10))
fact10
```

[46]:
```
(1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
```

An implementation without recursion can be created using NumPy's `prod()` function which computes the product of a sequence of numbers:

[47]:
```
import numpy as np
fact10 =  tuple(np.prod(range(1,n+1)) for n in range(10))
fact10
```

[47]:
```
(1.0, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
```

Notice that the first element of this sequence is a floating-point value 1.0, while the remaining elements are integers. Why is that? Examine the argument passed to `np.prod()` for `n=0`:

[48]:
```
n = 0
# We have to embed range() in an expression that forces the Python
# interpreter to actually expand the range object, such as a tuple().
tuple(range(1,n+1))
```

[48]:
```
()
```

As you see, for `n=0` this is an empty container without elements. The mathematical convention is that the product over an empty set is $\prod_{i \in \emptyset} = 1$, and this is exactly what `np.prod()` returns. However, by default NumPy creates floating-point values, and so the return value is 1.0, not 1.

You can get around this by explicitly specifying the data type using the `dtype` argument, which is accepted by many NumPy functions.

[49]:
```
import numpy as np
# Force result to be of integer type
fact10 =  tuple(np.prod(range(1,n+1), dtype=int) for n in range(10))
fact10
```

[49]:
```
(1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
```

Alternatively, we can use `np.arange()` instead of `range()` as the former by default returns integer arrays, even if they are empty:

[50]:
```
import numpy as np
# Force result to be of integer type
fact10 =  tuple(np.prod(np.arange(1,n+1)) for n in range(10))
fact10
```

[50]:
```
(1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
```

Finally, you of course would not need to implement the factorial function yourself, as there is one in the `math` module shipped with Python:

[51]:
```
import math
fact10 = tuple(math.factorial(n) for n in range(10))
fact10
```

[51]:
```
(1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
```

1.4.5 Solution for exercise 5

Below you find a simple implementation of a bisection algorithm. This function does not perform any error checking and assumes that the initial bracket $[a, b]$ actually contains a root, and that the values $f(a)$ and $f(b)$ have opposite signs.

Note that we impose two termination criteria, and the algorithm will end successfully whenever one of them is satisfied:

1. The function value is sufficiently close to zero, i.e., $|f(x_0)| < \epsilon$ for some small $\epsilon > 0$.
2. The bracket is sufficiently small, i.e., $|b - a| < \epsilon_x$, again for some small $\epsilon_x > 0$

This is standard practice in numerical optimisation since we don't want the algorithm to continue unnecessarily if the desired degree of precision was achieved.

We specify the termination tolerance as optional arguments `tol` and `xtol` with sensible defaults. We also add the maximum permissible number of iterations as an optional argument `maxiter`.

[52]:
```
def bisect(f, a, b, tol=1.0e-6, xtol=1.0e-6, maxiter=100):

    for iteration in range(maxiter):
        # Compute candidate value as midpoint between a and b
        mid = (a + b) / 2.0
        if abs(b-a) < xtol:
            # Remaining interval is too small
            break

        fmid = f(mid)

        if abs(fmid) < tol:
            # function value is close enough to zero
            break

        print(f'Iteration {iteration}: f(mid) = {fmid:.4e}')
        if fmid*f(b) > 0.0:
            # f(mid) and f(b) have the same sign, update upper bound b
```

```
            print(f'  Updating upper bound to {mid:.8f}')
            b = mid
        else:
            # f(mid) and f(a) have the same sign, or at least one of
            # them is zero.
            print(f'  Updating lower bound to {mid:.8f}')
            a = mid

    return mid

# Compute root of f(x) = x^2 - 4 on the interval [-3, 0]
# We pass the function f as the first argument, and use a lambda expression
# to define the function directly in the call.
x0 = bisect(lambda x: x**2.0 - 4.0, -3.0, 0.0)

# Print root. The true value is -2.0
x0
```

```
Iteration 0: f(mid) = -1.7500e+00
  Updating upper bound to -1.50000000
Iteration 1: f(mid) = 1.0625e+00
  Updating lower bound to -2.25000000
Iteration 2: f(mid) = -4.8438e-01
  Updating upper bound to -1.87500000
Iteration 3: f(mid) = 2.5391e-01
  Updating lower bound to -2.06250000
Iteration 4: f(mid) = -1.2402e-01
  Updating upper bound to -1.96875000
Iteration 5: f(mid) = 6.2744e-02
  Updating lower bound to -2.01562500
Iteration 6: f(mid) = -3.1189e-02
  Updating upper bound to -1.99218750
Iteration 7: f(mid) = 1.5640e-02
  Updating lower bound to -2.00390625
Iteration 8: f(mid) = -7.8087e-03
  Updating upper bound to -1.99804688
Iteration 9: f(mid) = 3.9072e-03
  Updating lower bound to -2.00097656
Iteration 10: f(mid) = -1.9529e-03
  Updating upper bound to -1.99951172
Iteration 11: f(mid) = 9.7662e-04
  Updating lower bound to -2.00024414
Iteration 12: f(mid) = -4.8827e-04
  Updating upper bound to -1.99987793
Iteration 13: f(mid) = 2.4414e-04
  Updating lower bound to -2.00006104
Iteration 14: f(mid) = -1.2207e-04
  Updating upper bound to -1.99996948
Iteration 15: f(mid) = 6.1035e-05
  Updating lower bound to -2.00001526
Iteration 16: f(mid) = -3.0518e-05
  Updating upper bound to -1.99999237
Iteration 17: f(mid) = 1.5259e-05
  Updating lower bound to -2.00000381
Iteration 18: f(mid) = -7.6294e-06
  Updating upper bound to -1.99999809
Iteration 19: f(mid) = 3.8147e-06
  Updating lower bound to -2.00000095
Iteration 20: f(mid) = -1.9073e-06
  Updating upper bound to -1.99999952
```

[52]: -2.000000238418579