16. Functions

Summary:

- Defining functions;
- Passing mutable objects as parameters;
- Global and local variables;
- Lambda expressions;
- Generators.

1 Basic function definitions

Basic syntax:

```
def func_name(arg_1, arg_2, ...):
    body
    return ...
```

Example, factorial calculation:

```
def fact(n):
'''Return the factorial of the given number n''' # docstring
    r = 1
    while n > 0:
        r *= n
        n -= 1
    return r
```

The `docstring` can be accessed by `.__doc__`:

```
fact.__doc__
> 'Return the factorial of the given number n'
```

Note: in some languages a function that does not return a value is called a procedure. Although you can in Python write functions that do not have a return statement, they are not really procedures. **All Python procedures are functions:** if no explicit return is executed in the procedure body, then the value None is returned. **This is advised by PEP8 to return None explicitly in such cases.**

2 Function arguments passing

Summary:

- Positional arguments;
- Keyword-passed arguments;
- Indefinite number of positional arguments;
- Indefinite number of keyword-passed arguments.

2.1 Positional arguments

The simplest way to **pass arguments is by position**.

1. In the function definition you specify variable names for each argument;
2. When the function is called, arguments used in the calling are matched to the function's parameters **by their order**.

```
def power(x, y):
    """Power operation"""
    r = 1 # local argument
    while y > 0:
        r *= x
        y -= 1
    return r
```

```
power(9,2) --> 81
power(2,9) --> 512
```

Default arguments

```
def fun(arg_1, arg_2=default_2, arg_3=default_3, . . .)
    ...
```

By this, the default arguments can be ignored in function call -> **readability**.

2.2 Keyword passing

You can also **pass** arguments into a function **using the name of the corresponding function parameter**, rather than its position. Because the arguments passed to function in its call are named, their o**rder is irrelevant**.

Keyword passing combined with the default argumenting, is useful when defining functions with large numbers of possible arguments, **most of which have common defaults**:

```
def list_file_info(size=False, create_date=False, mod_date=False, ...):
    # ...get files...
    if size:
        # code to get file sizes goes here
    if create_date:
        # code to get create dates goes here
```

This function call:

```
fileinfo = list_file_info(size=True)
```

2.3 Indefinite number of positional arguments

$*$ prefix before a named argument makes all excessing non-keyword arguments **to be collected in a tuple with the corresponding name**:

```
def maximum(*numbers):
    """ Find max among numbers"""
    if len(numbers) == 0:
       return None
    else:
        maxnum = numbers[0]
        for n in numbers[1:]:
            if n > maxnum:
            maxnum = n
    return maxnum
```

```
maximum(1,2) --> 2
maximum(1, 2, 76, 2, 3, 4, 8, 8, 9) --> 76
```

2.4 Indefinite number of keyworded arguments

** prefix before a named argument will make excessing keyword-passing arguments **to be collected into a dictionary with a corresponding name**:

```
def example_fun(x, y, **other):
print(f'x: {x}, y: {y}, keys in "other": {list(other.keys())}')
```

```
example_fun(2, 3, foo=3, bee=5)
> x: 2, y: 3, keys in "other": ['foo', 'bee']
```

3 Mutable objects as arguments

Arguments are passed in by object reference. The parameter name becomes a new reference to the passed object.

For immutable objects (e.g. tuples, strings, and numbers), what is done with an argument has no effect outside the function. **But any change made to the mutable object will change what the argument is referencing outside the function**:

```
def func(n, list1, list2):
    n = n + 1  # set to refer to new local object
    list1.append(3)
    list2 = [4, 5, 6]  # defined another object of same name but in the local namespace
    return None
```

```
x = 5       # isn't changed outside function because immutable
y = [1, 2]  # actual list was changed
z = [4, 5]  # was not changed

func(x, y, z)
print(x, y, z)
> 5 [1, 2, 3] [4, 5]
```

4 Local, nonlocal, and global variables

Local variables

Consider the factorial function from the previous sections:

```
def fact(n):
    '''Return the factorial of the given number n'''
     r = 1          # local variable
     while n > 0:
          r *= n
          n -= 1    # local variable
     return r
```

Variables `r` and `n` are **local** to any particular call of the function. **Changes to them made inside the function have no effect on any variables outside it**.

global a

You can explicitly make a variable **global**, using the `global` statement when declaring. **Global variables can be accessed and changed by the function since it exist outside the function**:

```
def fun():
    global a
    a = 1
    b = 2
```

```
a = 'one'
b = 'two'

fun()
print(a, b)
> 1 two
```

nonlocal a

Nonlocal variable are **used in nested functions** whose local scope is not defined. This means, the variable can be neither in the local nor the global scope.

Compare **without** nonlocal:

```
x = 0
def outer():
    x = 1
    def inner():
        x = 2
        print("inner:", x)
    inner()
    print("outer:", x)

print("globl:", x)
outer()
```

```
globl: 0
inner: 2
outer: 1
```

Now **use nonlocal** statement for inner x:

```
x = 0
def outer():
    x = 1
    def inner():
        nonlocal x
        x = 2
        print("inner:", x)
    inner()
    print("outer:", x)

print("globl:", x)
outer()
```

```
globl: 0
inner: 2
outer: 2
```

Also use **global** statement for inner x:

```
x = 0
def outer():
        x = 1
        def inner():
            global x
            x = 2
            print("inner:", x)
        inner()
        print("outer:", x)

print("globl:", x)
outer()
```

```
globl: 2
inner: 1
outer: 2
```

5 Assigning functions to variables: replacing switch-case

Functions can be assigned, like other Python objects, **to variables**.

Placing function names in list is a common pattern in situations where different functions need to be selected based on a string value, and in many cases it takes the place of the switch structure:

```
def f_to_kelvin(degrees_f):
    return 273.15 + (degrees_f - 32) * 5 / 9

def c_to_kelvin(degrees_c):
    return 273.15 + degrees_c

t = {'FtoK': f_to_kelvin, 'CtoK': c_to_kelvin}
```

```
t['FtoK'](100) --> 310.92777777777775
t['CtoK'](0) --> 273.15
```

6 Lambda expressions

lambda expressions are **(anonymous) little functions** that you can quickly define inline:

```
l = lambda arg1, arg2, ... : expression
```

Example:

```
t2 = {'FtoK': lambda deg_f: 273.15 + (deg_f - 32) * 5/9,
      'CtoK': lambda deg_c: 273.15 + deg_c}

t2['FtoK'](32)  -- > 273.15
```

7 Generators

Generator functions allow to **declare a function that behaves as iterator**.

An **iterator** is an object that can be iterated (looped) upon. It is used to abstract a container of data to make it behave like an iterable object. Iterators don't compute the value of each item when instantiated.

Generators are **functions that can be paused and resumed on the fly, returning an object that can be iterated over**. Unlike lists, they are **lazy** and thus **produce items one at a time and only when asked**. So they are much more **memory efficient when dealing with large datasets.**

The difference from conventional function is that it **saves the state of the function**. The next time the function is called, execution continues from where it left off, with the same variable values it had before yielding.

7.1 Generator functions

yield a

Define a function as you normally would but use the `yield` statement instead of **`return`**, indicating to the interpreter that this function should be treated as an iterator.

The `yield` statement **pauses the function and saves the local state so that it can be resumed right where it left off**:

```
def countdown(num):
    print('Starting')
    while num > 0:
        yield num
        num -= 1
```

When called, this function **returns a *generator object* – calling the function does not execute it**!

```
val = countdown(2)
> <generator object countdown at 0x00000252715047C8>
```

next( )

Generator objects executed when **next() over it** is called:

```
next(val)
> Starting
> 2
next(val)
> 1
next(val)
> StopIteration:
```

in

You can also use generator functions with in **to check if a particular value is in the series that the generator produces:**

```
2 in countdown(3)
> Starting
> True
```

7.2 Generator expressions

```
(expression for i in iterable)
```

Generator expressions are drastically faster **when the size of your data is larger than the available memory**.

Generator expressions **can run slower than list comprehensions** (unless you run out of memory, of course) i.e. they are memory efficient.

Similar to list comprehensions, generator expressions can also be written in the same manner except they **return a generator object rather than a list**:

```
gen_obj = (x/2 for x in range(3))

for val in gen_obj:
    print(val)

> 0.0
> 0.5
> 1.0
> Wall time: 0 ns
```

Be **careful not to mix up the syntax with a list comprehension** expression [ ] vs **( )** – since **generator** expressions **can run slower** than list comprehensions (unless you run out of memory, of course)

7.3 Generator usecase

Generators are perfect for **reading a large number of large files since they yield out data a single chunk at a time irrespective of the size of the input stream**. They can also result in cleaner code by decoupling the iteration process into smaller components. *Example:*

This function loops through a set of files in the specified directory. It opens each file and then loops through each line to test for the pattern match.

```
def emit_lines(pattern=None):                                    # find pattern in files
    lines = []                                                   # preallocate space for lines
    for dir_path, dir_names, file_names in os.walk('test/'):     # go through the files
        for file_name in file_names:                             # for each file in the folder...
            if file_name.endswith('.py'):                        # if this is .py file
            for line in open(os.path.join(dir_path, file_name)): # for each line in opened file
                if pattern in line:                                 # if pattern found
                   lines.append(line)                            # append line in preallocated list
    return lines
```

This works well with the small number of small files. `open()` function is quite efficient but what if we are dealing with quite large files? And what if our matches far exceeds the available memory on our machine?

So, instead of running out of space (large lists) and time (nearly infinite amount of data stream) when processing large amounts of data, **generators** are the ideal things to use, as they **yield out data one time at a time** (instead of creating intermediate lists).

We divided our whole process into three different components:

- Generating set of filenames;
- Generating all lines from all files;
- Filtering out lines on the basis of pattern matching.

Essentially we create **nested generators**:

```
def generate_filenames():  # generator of opened files
    """ generates a sequence of opened files matching a specific extension"""
    for dir_path, dir_names, file_names in os.walk('test/'):
        for file_name in file_names:
            if file_name.endswith('.py'):
               yield open(os.path.join(dir_path, file_name))

def cat_files(files): # generator of lines
    """ takes in an iterable of filenames"""
    for fname in files:
        for line in fname:
            yield line

def grep_files(lines, pattern=None):  # generator of filtered lines
    """ takes in an iterable of lines """
    for line in lines:
        if pattern in line:
           yield line

# construct a nested generator:
py_files = generate_filenames()
py_lines = cat_files(py_files)
filtered_lines = grep_files(py_lines, 'pattern_line_example')

# then just print the result from the defined above composite generator:
for line in filtered_lines:
    print(line)
```

We do not use any extra variables to form the list of lines, instead we create **a pipeline which feeds its components via the iteration process one item at a time**.