Week 6: Functions

We have used functions already in exercises and projects, but we haven't studied how they work in detail. That ends today!

Exercises

DocTests and PyTest

This exercise isn't an exercise as such, but rather a meta-exercise... it is time to learn a little bit about how you should test your code.

The first way to test your code is what you have already been doing. You write some code, you run it, and then you check if the results make any sense. Sometimes they don't, and you have to fiddle with it some more. Sometimes you are lucky and things seem to be working — for those few examples you tried it on, at least.

This is a very manual way to test your code, and it will only ever test a few examples. You usually cannot test thoroughly if you have to type in cases yourself. Ideally, you want to write code that tests your functionality, since that can explore more of the problem domain than you can manually.

Ideally, you want code for testing your code that is run every time you fiddle with it, so it can warn you if you have broken anything.

There are multiple levels of testing, depending on how much you test and when you test, but the most basic is called *unit testing* and that is when you have (preferably automated) tests for all "units". What a unit is varies depending on who you ask and what language you use, but in Python a unit would be something like a function or a class.

Most languages have multiple systems for unit testing — it is such an important part of software development — and Python is no exception. We have used two kinds in the class so far: `doctest` and `pytest`. From now on, I want you to write your own tests using these.

The `doctest` module is the simplest. It uses documentation strings in functions to test if what the documentation says is also true. If behaviour of a function doesn't match the documentation, then something is definitely wrong. If, in a function's documentation string, you write `>>>` and a function call

```
>>> fact(4)
```

and then the expected output on the line after that, so

```
>>> fact(4)
24
```

then `doctest` will check that this is what happens when you invoke it.

You can run `doctest` on a file with

```
> python3 -m doctest file.py
```

Since `doctest` is tied to function documentation, and since no one wants to read documentation that contains pages and pages of test code, it is a little limited. It is great for making sure that the documented behaviour is also what happens, but for better tests you can use the `pytest` module. if it is not already installed, you can install it with

```
> pip install -U pytest
```

The simplest way to use this — and we don't need to make it complicated—is to write your tests in files whose names start with `test_` and in functions whose names start with `test_` as well.

For example, we could write this function

```
def test_fact():
    assert fact(4) == 42
```

and put it in a file `test_fact.py`. If you put all your test files in a directory, say `testfolder`, you can run them all with

```
> python3 -m pytest testfolder
```

What I would like you to do from now on is to write `doctest` comments for the functions you write and preferably at least one test function for `pytest` for each function you write. You will find that most repositories in this course will have `test_` files in the `src/` directory that you should fill out with your own tests.

Simple functions

If you would like template code for this exercise, you can get it here.

Exercise: Write a function that takes three numbers as arguments and return their product.

Exercise: Change the function so it only takes one argument, get another from the global scope, and assigns to a local variable for the last value.

Exercise: Write a function that takes two lists as its input and return the longest of the two.

Exercise: The distance between two points, `(x1,y1)` and `(x2,y2)` is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2)}$. It follows from Pythagora's Theorem (as you should be able to see using a small drawing). Write a function that computes the distance between two functions. You can use the `sqrt()` function to get the square root, but you need to import it from the `math` module first:

```
from math import sqrt
```

Alternatively, you can simply use the `**` operator to get the square root. `x**2` is x^2, and `x**0.5` is the square root of `x`, $x^{1/2}$.

DNA

k-mer exercises

If you want a template to work from for these execises, you can get it here.

A k-mer is a substring of length k of some string s. For example, the k-mers for k = 3 of

```
agtagtcg
```

are

```
agt
gta
tag
agt
gtc
tcg
```

Exercise: Write a function, `kmers`, which returns a list of all k-mers in a given string for a specific k:

Exercise: Often, we're only interested in unique k-mers. Write a function, `unique_kmers`, which only returns unique k-mers.

Hint: Can you use a *set* to make this easier? Sets will filter elements so they are all unique. Sets, however, do not preserve the order in which you insert them, so if you want that (and that is up to you), they might not be the right solution. *Dictionaries*, on the other hand, do preserve insertion order, so they might give you a way to do it.

Exercise: Now try to write a function that returns a list of all k-mers paired together with the number of occurrences of that k-mer in the string. Consider it a bonus if you return the list sorted according to the number of occurrences.

Codon translation

Codon translation is the process of translating a sequence of nucleotides (for example a DNA sequence) into a sequence of amino acids (the building blocks of proteins). A codon (at least for life as we know it) is a sequence of three nucleotides, and each codon corresponds to an amino acid. For example, the codon `ATG` corresponds to the amino acid `Methionine` (Met), and the codon `TAA` corresponds to the amino acid `Stop` (Stop). The codon `ATG` is also the start codon, and it is the codon that starts the translation process. Further explaination can be found in the `readme.md` file in the template repository.

Exercise: Complete the exercise described in the template repository about codon translation.

More OPTIONAL exercises on functions

There are many aspects about functions that we don't cover in this course that might be nice to at least be familiar with. This section contains some exercises that you can do if you want to learn more about functions. **They are not required for the course and we will not go through them in class**, but they might be useful to you in the future.

Exceptions

Exercise: A DNA string consists of A, C, G, and T letters. Write a function that counts how many times each of the letters occur in a string, but raises an exception if there is a letter that isn't one of these four.

Exercise: Consider this example, which I admit is a little complicated, but I will explain it below:

```
def raises_error(x):
 if x < 0:
  raise Exception("Negative", x)
 if x > 0:
  raise Exception("Positive", x)
 return 42

def f(x):
 return raises_error(x)

def g(x):
 try:
  print(f(x))
 except Exception as e:
  if e.args[0] == "Negative":
   print("g:", e.args[0])
   return f(0)
  else:
   raise
```

We have a function, `raise_error()`, that will raise an exception if its input is either negative or positive, but return 42 if its input is 0. The function `f()` just calls `raise_error()` and cannot handle any exceptions. So if `raises_error()` throws an exception, then `f()` will simply propagate it to its caller. Function `g()` calls `f()`, but inside a `try/except`. It will print the result of `f(x)` if `f()` returns normally, but if `f()` propagates an exception, we jump to `g()`'s `except` block. Here, we name the exception `e`, and we check if it was a "Negative" error we got. If so, we print a message and call `f(0)` and returns the value (which will be 42). If `e` is a "Positive" error, we re-raise the exception using `raise` with no argument.

Now, try running this code:

```
try:
 print(g(-1))
 print(g(1))
except Exception as e:
 print("Outer", e.args[0])
```

The first time we call `g()`, it will get an exception from `raises_error()` through `f()`. It can handle this exception, so it prints "g: Negative" and returns `f(0)`, which is 42, and `print(g(-1))` prints 42. The second time we call `g()`, `raises_error()` raises an exception again, but this time `g()` cannot handle it, so it propagates it to the global `try/except`, where the `except` block handles the exception and prints "Outer Positive".

Exercise: Go through this code carefully and make sure you understand what is happening. Make a drawing of the function calls for the two calls to `g()` and how exceptions propagate from `raises_error()`.

Parameterised insertion sort

Consider insertion sort:

```
def insertion_sort(x):
 for i in range(1,len(x)):
  j = i
  while j > 0 and x[j-1] > x[j]:
   x[j-1], x[j] = x[j], x[j-1]
   j -= 1
 return x
```

It sorts its input in increasing order. What if we want to sort in decreasing order as well? We can parameterise the comparison `x[j-1] > x[j]`, and get what we want:

```
def insertion_sort(x, greater_than):
 for i in range(1,len(x)):
  j = i
  while j > 0 and greater_than(x[j-1], x[j]):
   x[j-1], x[j] = x[j], x[j-1]
   j -= 1
 return x

def greater(x, y):
 return x > y
insertion_sort(x, greater)

def smaller(x, y):
```

```
 return y > x
insertion_sort(x, smaller)
```

We use a function to compare two elements, and we use two functions for the comparison, one that says that `x` is larger than `y`, so we will sort in increasing order, and one that says that `x` is smaller than `y`, and that will sort the elements in decreasing order.

Exercise: Go through the code and make sure you understand why it sorts in increasing and decreasing order.

Accumulate and Reduce

There is a classic function in functional programming called `reduce()` or `fold()`. You will find it in practically any programming language that supports functional programming, since it is a generalisation of many other functions that you would have to implement explicitly otherwise.

Consider a sequence `x = [x[0], x[1], ..., x[n-1]]` and a function `f` and define the sequence `y` by

```
y[0] = x[0]
y[1] = f(y[0],x[1])
y[2] = f(y[1],x[2])
y[3] = f(y[2],x[3])
...
y[n-1] = f(y[n-2],x[n-1])
```

i.e. `y[0]` is `x[0]` and every other `y` you get as `y[i] = f(y[i-1],x[i])`.

If you "reduce" `x` with function `f` it means that you compute `y[n-1]` which is

```
reduce(f,x) == f(f(f(...f(f(x[0],x[1]),x[2])...x[n-1])
```

If you want the intermediate values, so `y = [y[0], y[1], ..., y[n-1]]` you "accumulate" `x` with respect to `f`

```
accumulate(f,x) == [x[0], f(y[0],x[1]), ..., f(y[n-2],x[n-1])]
```

These two are (of course) already implemented in Python

```
from itertools import accumulate
from functools import reduce
```

Those functions are more general than we will attempt here (and they are the ones you should use if you find yourself in need of this functionality in the future), but just for fun, let's try to implement them.

To simplify things, we will assume that `len(x) >= 2` (since otherwise, we haven't define what the result might be). Could you think of other alternatives?

Exercise: Implement the `reduce()` function and use it to compute the sum and product of a list of numbers.

Exercise: Implement the `accumulate()` function to see the intermediate results as well.

You can start the exercise from this template repository.