Genbrug er guld!

This session is about reuse. Reuse of code. But let's first try to look at a metaphor.

Enacting Programs...

Script A - Jens, please find Viktoria's telephone number.

Helge talks to Jens

> **Helge**: Can you please look for Viktoria's telephone number in your notebook?
>
> *Jens is searching intensely in his notebook.*
>
> **Helge**: If you did not find her telephone number there then search in your mail inbox.
>
> *Jens is searching in his mail inbox getting a bit bored.*
>
> **Helge**: If you did not find her telephone number in you mail inbox then please search on ITU's intranet.
>
> *Jens is searching on ITU's intranet and gets annoyed by the task. But finally, there he finds what he was searching for.*
>
> **Helge**: Once you found number, please write it on a post-it.
>
> **Helge**: Please give me the post-it with the telephone number now.

Helge is happy and can call Viktoria concerning the course registrations.

Script B - Jens, please find Peter's telephone number.

Helge talks to Jens

> **Helge**: Can you please look for Peter's telephone number in your notebook?
>
> *Jens is searching intensely in his notebook.*
>
> **Helge**: If you did not find his telephone number there then search in your mail inbox.
>
> *Jens is searching in his mail inbox getting a bit bored.*
>
> **Helge**: If you did not find his telephone number in you mail inbox then please search on ITU's intranet.
>
> *Jens is searching on ITU's intranet and gets annoyed by the task. But finally, there he finds what he was searching for.*
>
> **Helge**: Once you found the number, please write it on a post-it.
>
> **Helge**: Please give me the post-it with the telephone number now.

Helge is happy and can call Peter concerning the the fall courses.

Pyha, that was quite repetitive

Let's encapsulate a sequence of statements (instructions) so that the script gets more terse and Helge has to talk less.

Script C - Jens, please find the Pizzaria's telephone number.

Helge talks to Jens

Pretext:

> **Helge**: You remember the steps for searching a telephone number right? I also just wrote them down for you on this sheet of paper.
>
> *Jens nods and takes over the note from Helge*

Main act:

> **Helge**: Can you please find me the telephone number from the 'Pizzaria La Fiorita'?
>
> *Jens starts searching according to the script on the notes. After some time, he is done searching and hands over the result of his search -the pizzaria's telephone number- to Helge.*

Helge is happy and can call La Fiorita to order lunch.

Reflection

- Discuss with your neighbors, what was going on in the three plays?
- When we call the encapsulation of statements a *function*, what were the arguments to the function in Script C?
- What did the function in Script C return?

Structured programming

Structured programming partly solves the complexity problem. It basically does it by delegation.

Instead of writing all code in one place we can put code into *structures* that have certain responsibilities.

These *structures* are also called *functions*. They are tasked with solving a single problem.

Creating a Function

You have actually already used functions. Both `print()` and `len()` are functions.

Python provides several built-in functions like these, but you can also write your own functions. Here is how:

In [2]:

```
def print_sentence():
    print('Call me Ishmael.')
```

In [3]:

```
print_sentence()
```

```
Call me Ishmael.
```

In [4]:

```
print_sentence()
```

```
Call me Ishmael.
```

In [8]:

```
print_your_something()
```

```
These are not the droids you are looking for!
```

In [5]:

```
def print_your_something():
    print('These are not the droids you are looking for!')


print_your_something()
```

```
These are not the droids you are looking for!
```

Try it out!

```
def print_something():
    ...


print_something()
```

Functions are tasks

If you need to perform that task multiple times throughout your program, you do not need to type all the code for the same task again and again; you just call the function dedicated to handling that task.

Your "**call**" tells Python to run the code inside the function. You will find that using functions makes your programs easier to write, read, test, and x.

In [ ]:

```
def print_something():                      # <--- Function definition
    print('Conchita Wurst is the best!')    # <--- Function body


print_something()                           # <--- Function call
```

In [ ]:

```
def print_something():                      # <--- Function definition
    print('Conchita Wurst is the best!')    # <--- Function body


print_something()                           # <--- Function call
print_something()                           # <--- Function call
```

How to create a function

```
def print_sentence():
    fst_sentence = 'Call me Ishmael.'
    print(fst_sentence)
```

This example shows the simplest structure of a function. It contains a **function definition** and a **function body**:

- The first line uses the keyword `def` to inform Python, that you are defining a function. This is the *function definition*, which tells the interpreter the name of the function and, if applicable, what kind of information the function needs to do its job. The parentheses hold that information, the *arguments*. In this case, the name of the function is `print_sentence()`, and it needs no information to do its job, so its parentheses are empty, there are no arguments. (Even so, the parentheses are required.) Finally, the definition ends in a colon.
- Any indented lines that follow `def print_sentence():` make up the *body of the function*.

The lines:

```
fst_sentence = 'Call me Ishmael.'
print(fst_sentence)
```

are the only lines of actual code in the body of this function, so `print_sentence()` has just one job, it prints the first sentence of Moby Dick.

When you want to use this function, you call it. A *function call* tells Python to execute the code in the function. To call a function, you write the name of the function, followed by any necessary arguments in parentheses. Because no information is needed here, calling our function is as simple as entering `print_sentence()`.

Functions return values

Functions **always** return something! Sometimes it is just *nothing* (value `None`) in Python.

In [10]:

```
def print_sentence():
    print('Call me Ishmael.')


# print_sentence()
result = print_sentence()
print(result)
```

```
Call me Ishmael.
None
```

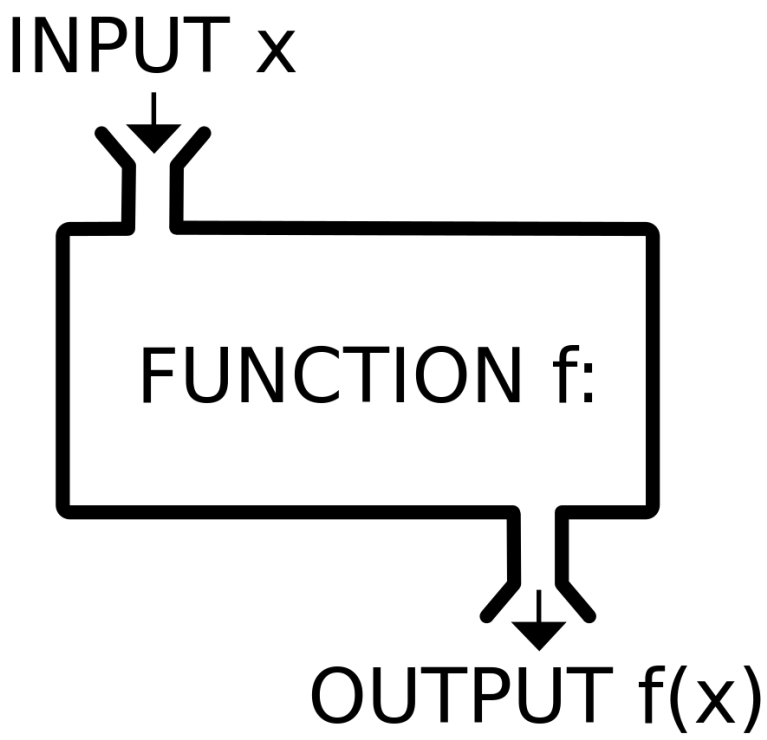
That is, assign the result of a function call only to a variable when you need it. You surely do not need the `None` value, i.e., nothing.

An adaptive function

The function above was pretty boring. It never changed behavior.

We can actually put something into the function, so it adapts to what we need:

Functions as "black boxes"



In [19]:

```
def extremely_difficult_calculation(input_number):
    result = input_number + 1  # Look at how difficult that was!
    return result


data = 1000
res = extremely_difficult_calculation(data)
print_return = print(res)
print(print_return)
```

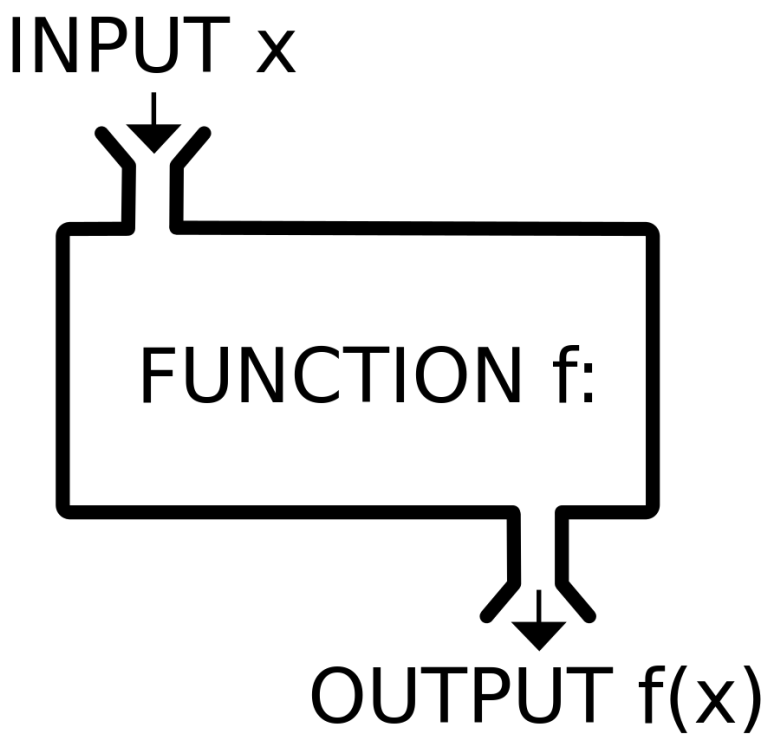
```
1001
None
```

Functions return values

Functions are delegations of code. You *delegate* work to the function, so you don't have to do it.

```
def extremely_difficult_calculation(input_number):
    result = input_number + 1  # Look at how difficult that was!
    return result
```

Functions are reusable black boxes



What happens above is that you create a function that takes an input and `return` s the output?

Modify the code below to no longer *print* the name, but *return* it?

```
def return_sentence():
    fst_sentence = 'Call me Ishmael.'
    print(fst_sentence)
```

Now try to call your function. What happens? And did you expect that?

Value assignment

These two pieces of code are 100% equivalent.

```
my_var = 'Call me Ishmael.'
```

```
my_var = return_sentence()
```

An adaptive function

The function above was pretty boring. It never changed behaviour: the turtle kept doing the same over and over again.

Just like a program with no `input` .

In [8]:

```
def modify_sentence(name):
    fst_sentence = 'Call me ' + name + '.'
    return fst_sentence
```

In [ ]:

```
modify_sentence('Ahab')
```

In [ ]:

```
modify_sentence('Michelle')
```

In [ ]:

```
def modify_sentence(name):
    fst_sentence = 'Call me ' + name + '.'
    return fst_sentence
```

In [ ]:

```
modify_sentence(23456)
```

Passing Information to a Function via Arguments

```
def modify_sentence(name):
    ...
```

Arguments and Parameters

In the above example, `modify_sentence(name)` requires a value for the variable `name`. Once we called the function and gave it the information -a person's name-, it can now do something with that name.

In practice `name` becomes a `variable` inside the function. But because it is required for the function to work, it is called a **parameter**: it is a piece of information the function *needs* to do its job.

In [ ]:

```
def modify_sentence(name):
    fst_sentence = 'Call me ' + name + '.'
    return fst_sentence
```

In [ ]:

```
modify_sentence('Ahab')
```

The value `'Ahab'` in `modify_sentence('Ahab')` is an example of an argument. An argument is a piece of information that is passed from a function call to a function. When we call the function, we place the value we want the function to work with in parentheses. In this case the argument `'Ahab'` was passed to the function `modify_sentence(name)`, and the value was stored in the parameter `name`.

Note, people sometimes speak of arguments and parameters interchangeably.

Argument Errors

When you start to use functions, do not be surprised if you encounter errors about unmatched arguments. Unmatched arguments occur when you provide fewer or more arguments than a function needs to do its work.

In [21]:

```
def apply_division(dividend, divisor):
    result = dividend / divisor
    print(result)


apply_division(10, 2)
```

```
5.0
```

Can you make this code divide the number 7 with the number 3: `7 / 3` ?

In [ ]:

```
def apply_division(dividend, divisor):
    result = dividend / divisor
    print(result)


apply_division()
```

Turtle drawing with functions

Type this into Mu:

```
from turtle import forward, turn


def forward_and_turn(angle):
    forward(200)
    left(angle)


forward_and_turn(90)
```

What happens if you call that function one more time? And again? Run it through the debugger.

Cookie function

In a previous session you wrote a piece of code that printed out how much you enjoyed cookie dough.

- Can you turn the following code into a function called `cookie_likeness` ? What should the argument be?

```
data = input('How much do you like cookie dough?')
data = int(data)
print('You ' + 'really ' * data + 'like cookie dough')
```

In [ ]:

```
def cookie_likeness(data):
    message = 'I ' + 'really ' * data + ' like cookie dough'
    return message


cookie_likeness(4)
```

How NASA does programming

One advantage of functions is the way they separate blocks of code from your main program. By using descriptive names for your functions, your main program will be much easier to follow.

Organisations like NASA take things like naming and structure very very serious. Bad programming decisions are causing deaths and vast productivity losses daily. They wrote [10 rules for developing safety-critical code (http://pixelscommander.com/wp-content/uploads/2014/12/P10.pdf)](http://pixelscommander.com/wp-content/uploads/2014/12/P10.pdf).

Bad example: [Mariner 1 (https://en.wikipedia.org/wiki/Mariner_1)](https://en.wikipedia.org/wiki/Mariner_1)

The fear of technical debt

NASA has good reason to do this. While code gets increasingly complicated, your brain does not.

Think about it this way: Every time you write a line of code, your program gets more complicated. Essentially that is a *debt* you will have to pay later when you want new features.

> The cost of never paying down this technical debt is clear; eventually the cost to deliver functionality will become so slow that it is easy for a well-designed competitive software product to overtake the badly-designed software in terms of features.
>
> - Junade Ali, Mastering PHP Design Patterns

Modules

You can organise your program further by storing your functions in a separate file called a module.

A module is a collection of functions that do something specific. There is for instance a module called `os` that contains functions related to your computer (operating system).

Modules can be imported in Python using the `import` keyword:

```
import os
```

An import statement tells Python to make the code in a module available in the currently running program file.

Storing your functions in a separate file allows you to hide the details of your program's code and focus on its higher-level logic. It also allows you to reuse functions in many different programs.

When you store your functions in separate files, you can share those files with other programmers without having to share your entire program. Knowing how to `import` functions also allows you to use libraries of functions that other programmers have written.

Importing a Module

To call a function from an imported module, enter the name of the module you imported followed by the name of the function separated by a dot.

In the following examples, we import entire modules, which makes every function from the module available in your program.

In [22]:

```
import os


cpus = os.cpu_count()
print(cpus)
```

```
12
```

Stand on the shoulders of giants (and 10'000+ man hours)

Storing your functions in a separate file allows you to hide the details of your program's code and focus on its higher-level logic. It also allows you to reuse functions in many different programs.

When you store your functions in separate modules, you can share those modules with other programmers without having to share your entire program.

Knowing how to `import` functions also **allows you to use libraries of functions that other programmers have written**.

Modules are Files

Modules are actually just files with a `.py` ending. So they all exist in a file: `__file__` . You can see that in Python if you write

```
print(__file__)
```

So if you write

```
import string
```

You actually just fetch the contents of the `string.py` file.

In [2]:

```
import string


print(string.__file__)
```

```
/usr/local/anaconda3/lib/python3.7/string.py
```

- The property `__file__` tells you, where a module is stored. For example, `/usr/local/anaconda3/lib/python3.7/string.py` is the path for where the `string.py` module is stored on Helge's computer and `C:\Users\vagrant\Anaconda3\lib\string.py` is tha path where that module is stored on Helge's Windows computer...

Type and run the following short program into the mu-editor

```
import string


print(string.__file__)
```

From the terminal, open the `string.py` file and inspect the first 50 lines of this module.

```
$ mu-editor <path/to/string.py>
```

Describe to your neighbor what you can see on the first 50 lines of `string.py` .

Module Aliases

Modules can be given aliases when imported:

```
import module_name as mn
```

Calling the functions via a module alias is more concise and allows you to focus on the descriptive names of the functions.

The function names, which clearly tell you what each function does, are more important to the readability of your code than using the full module name.

In [38]:

```
import random as r


r.choice([1, 2, 3, 4, 5, 6])
```

Out[38]:

```
4
```

In [14]:

```
import random as r


r.choice([1, 2, 3, 4, 5, 6])
```

Out[14]:

```
1
```

Talk through the code with your neighbour *before* running it.

When you execute it, does it do what you expected it to do?

In [ ]:

```
import random


def roll_dice(number_of_rolls):
    rolls = []
    for rolls in range(number_of_rolls):
        new_roll = random.choice([1,2,3,4,5,6])
        rolls.append(rolls)
    return rolls


number_of_times = input('How many times?')
roll_dice(number_of_times)
```

Importing Specific Functions

You can also import a specific function from a module. Here's the general syntax for this approach:

```
from module_name import function_name
```

You can import as many functions as you want from a module by separating each function's name with a comma:

```
from module_name import function_0, function_1, function_2
```

In [ ]:

```
from random import choice


choice([1, 2, 3, 4, 5, 6])
```

In [42]:

```
from random import choice, randrange
import random

end = choice([10, 20, 30, 40, 50, 60])
# Choose a random item from range(start, stop[, step]).
randrange(1, end)
```

Out[42]:

```
4
```

With this syntax, you do not need to use the dot notation when you call a function. Because we have explicitly imported the function in the `import` statement, we can call it by name when we use the function.

However, using the full name makes for more readable code, so it is better to use the normal form of the import statement.