Python

A quickstart into the key concepts of programming

Functions

functions

functions.ipynb

Functions

- A function is a named block of programming statements designed to perform a certain task.
- Wrap code up in a **function**, be able to repeatedly use it
  - A piece of code written to carry out a specified task.
  - This makes code easier to read, maintain and debug.
  - Code can be tested separately
- The output of the function is stated by “return”
- Programming functions (usually) take *arguments*.
  - The input comes in in parentheses after the function name:
  - The ordered sequence of variables is strictly speaking, called the *argument list* in the caller and the *parameter list* in the function definition.

Why functions?

- Name a group of statements, which makes your program easier to read. **D**on’t **R**epeat **Y**ourself (DRY principle)
- Functions make programs smaller by reusing the same code.
- Dividing programs into functions allow for debugging sections at a time. Easier maintenance of the code.
- Well defined functions can be used in other programming projects (generalization).

How to design a function?

1. Wishful thinking: Write the program as if the function already exists
2. Write a specification: Describe the inputs and output, including their types
   - No implementation yet!
3. Write tests: Example inputs and outputs
4. Write the function body (the implementation)
   - write your plan in words, pseudo-code
   - then translate to Python

- Source: https://courses.cs.washington.edu/courses/cse160/15sp/lectures/04-functions.pptx

Functions

- Skeleton / syntax

```
def myfunc(p1, p2, ..):
    “”” info “””
    statement 1
    statement 2
    …
    return statement
```

- The *def* keyword introduces the function.
- A colon *[:]* ends the introduction to the function.
- *myfunc* is the name of the function.
- *parameter* **list** inside ()
  - Empty parentheses when no parameters are used.
- All statements are indented
- Usually an (optional) `return` statement that can return the result to the calling program.

Function Naming rules

- Must start with a letter or an underscore(_)
- Should be lowercase.
- Can contain numbers.
- No spaces
- Case sensitive
- Can be any length (reasonably short), meaningful
- Two different functions can't have the same name
- Cannot use any of Python's keywords

Functions

- A function must be called, it does nothing by itself.
- The interpreter must see the function definition before it can be called.
  - Best practice is to put all the functions at the top of a file, right after the main docstring.
- End the function with a `return` statement if the function should output something.
  - Without the return statement, the function will return an object `None`.
  - Return multiple values in a tuple ()
- *File: function_1.py*

function_1.py

```
def my_func_1 ():
    """  this function writes hello  """
    print('Hello')
def my_func_2(a, b):
    """    parameters:
        a: first part of mathematical manip
        b: second part in mathematical manip
    """
    res = a + b
    return (res)
def my_func_3(a, b):
    """
    parameters:
        a: first part of mathematical manip
        b: second part in mathematical manip
    """
    res1 = a + b
    res2 = a - b
    return (res1, res2)
```

```
# main part, calling the functions
my_func_1()

my_func_2(5,6) # effect?
a = my_func_2(5,6)
print(type(a))
print(a)

b = my_func_3(5,6)
print(type(b))
print(b)
```

Functions: how to call a function

- Call a function by its name and pass the arguments
- Mechanism: call by object reference
  - pass immutable arguments like integers, strings or tuples to a function, the passing acts like call-by-value.
  - Pass mutable arguments (e.g. lists), consider two cases:
    - Elements of a list can be changed in place
    - If a new list is assigned to the name, the old list will not be affected.
- Argument lists in the caller and the callee must agree in *number* and *type*.
  - Default arguments
  - Required arguments
  - Keyword arguments
  - Variable number of arguments
- *File: function_argument_pass_1.py*

Optional and keyword arguments

- An argument may be assigned a default value in the parameter list
  - If not present in the calling argument list, the default value will be used
- Keyword arguments can be passed by keyword, not position. They must *follow* any positional arguments in the argument list.
- *File: function_optional_arguments.py*

```
def foo(val1, val2, val3, calcSum=True):
    # Calculate the sum
    if calcSum:
        return val1 + val2 - val3
    # Calculate the average instead
    else:
        return (val1 + val2 + val3) / 3

print(foo(10,20,30, True))
print(foo(10,20,30, False))
print(foo(10,20,30))

print(foo(val1=10,val3=20,val2=30,calcSum=True))
```

Returning from a function

- Values are returned by the `return` statement. It can return an item or an expression.
  - Return: immediate exit.
  - Python functions can return only one item but that item can be any object, in particular a tuple, list, or dictionary.
- A function may have multiple return statements but only one (the first encountered) will be executed.
- Functions always return something. If you do not specify a return value Python returns the special value `None`.

*args and **kwargs

- variable length argument lists
- *args is used to pass a non-keyworded, variable-length argument list
  - * before a variable means "*expand this as a sequence*"
  - Unpacking operator
- **kwargs is used to pass a keyworded, variable-length argument list.
  - ** before a variable means "*expand this as a dictionary*"
- Useful when:
  - Reduce code rewriting.
  - Reuse your code
- *File: function_arg_kwarg_1.py*
- Source: https://www.saltycrane.com/blog/2008/01/how-to-use-args-and-kwargs-in-python/
- https://realpython.com/python-kwargs-and-args/

Functions: scope

- Functions can be considered as a mini program
- Variables can be created inside functions
  - are considered local to that function.
  - they only exist within that function.
  - Objects outside the scope of the function will not be able to access that variable
  - To create a global variable inside a function, you can use the `global` keyword.
    - *Be careful with global variables*
- Variables that are only referenced inside a function are implicitly global. If a variable is assigned a value anywhere within the function's body, it's assumed to be a local unless explicitly declared as global.
- *File: function_scope_5.py*

Functions docstrings

- Docstrings describe what the function does. These descriptions serve as documentation for the function
- Function docstrings are placed after the function header and are placed in between triple quotation marks.
- The docstring must be indented to the level of the function body.

```
def hello():
    """
    Prints "Hello World".
    Returns:
       None
  """
  print("Hello World")
  return
print("The docstring of the 
function hello: " + 
hello.__doc__)
```

- *File: hello.py*
  help('hello')

Module

Stepping stones in programming projects:

- Start working with Python in notebook or on the interpreter
- Write longer programs that need to be executed multiple times: write scripts
- The programs grow in size, split it into several files for easier maintenance as well as reusability of the code.
  - The solution: Modules.
  - Define most used functions in a module and import it, instead of copying their definitions into different programs.
  - A module can be imported by another program to make use of its functionality.

Module

- Module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.
- Organize Python code logically. Grouping related code into a module makes the code easier to understand and use.
- For the Python interpreter to find the functions in an external file, the file must either be in the same directory as the program calling the function, or the file must be located underneath special directories designated by **sys.path**.
- *File: support_print.py (module containing the functions)*
- *File: support_print_use.py (program using the functions)*

Loading Modules: the `import` Statement

- Explicit module import : Explicit import of a module preserves the module's content in a namespace. The namespace is then used to refer to its contents with a "." between them.

```
import math
math.cos(math.pi)
```

- Explicit module import by alias
  - For long module names, use the "`import ... as ...`"

```
import numpy as np
```

- Explicit import of module contents: Sometimes rather than importing the module namespace, just like import a few particular items from the module, use "from ... import ..."

```
 from math import cos, pi
```

- Implicit import of module contents: it is sometimes useful to import the entirety of the module contents into the local namespace. This can be done with the "from ... import *"

```
from math import *
sin(pi) ** 2 + cos(pi) ** 2
```

  - This pattern should be used sparingly, if at all. The problem is that such imports can sometimes overwrite function names that you do not intend to overwrite, and the implicitness of the statement makes it difficult to determine what has changed.

https://nbviewer.jupyter.org/github/jakevdp/WhirlwindTourOfPython/blob/master/13-Modules-and-Packages.ipynb

Terminology

- Taken from: https://realpython.com/lessons/scripts-modules-packages-and-libraries/#:~:text=04%3A41%20Packages%20are%20a,Python%20scripts%20without%20any%20issues
- A **script** is a Python file that's intended to be run directly. When you run it, it should do something. This means that scripts will often contain code written outside the scope of any classes or functions.
- A **module** is a Python file that's intended to be imported into scripts or other modules. It often defines members like classes, functions, and variables intended to be used in other files that import it.
- A **package** is a *collection of related modules* that work together to provide certain functionality. These modules are contained within a folder and can be imported just like any other modules. This folder will often contain a special __init__ file that tells Python it's a package, potentially containing more modules nested within subfolders
- A **library** is an umbrella term that loosely means "a bundle of code." These can have tens or even hundreds of individual modules that can provide a wide range of functionality. Matplotlib is a plotting library. The Python Standard Library contains hundreds of modules for performing common tasks, like sending emails or reading JSON data. What's special about the Standard Library is that it comes bundled with your installation of Python, so you can use its modules without having to download them from anywhere.

Standard library

- Python's standard library contains many useful built-in modules, Python documentation (https://docs.python.org/3/library/)
- Batteries included
- Any of these can be imported with the `import` statement
  - os and sys: Tools for interfacing with the operating system, including navigating file directory structures and executing shell commands
  - math and cmath: Mathematical functions and operations on real and complex numbers
  - itertools: Tools for constructing and interacting with iterators and generators
  - functools: Tools that assist with functional programming
  - random: Tools for generating pseudorandom numbers
  - pickle: Tools for object persistence: saving objects to and loading objects from disk
  - json and csv: Tools for reading JSON-formatted and CSV-formatted files.
  - urllib: Tools for doing HTTP and other web requests.

https://nbviewer.jupyter.org/github/jakevdp/WhirlwindTourOfPython/blob/master/13-Modules-and-Packages.ipynb

name == main block

- 2 ways of executing code
  - As main process, execute the file as a Python Script (most common).
  - Being imported as a module.
- Python interpreter sets **__name__** depending on the way how the code is executed.
  - running the script directly, Python is going to assign "**__main__**" to **__name__**
  - **__name__** variable helps to check if the file is being run directly or if it has been imported.

name == main block

- The main block is only run when the file is run directly by Python. When the file is imported, the main block is not run.
- Usage: put the testing code in the main block, so that when you're importing the file, the functions can be used, but the testing is not seen
- *File: isVow_using.py*

Lambda expressions

- Lambda expressions create anonymous functions
- Must be expressible as a single expression (it has to return something)
  - Arithmetic operations like a+b and a**b
  - Function calls like sum(a,b)
  - A print statement like print(“Hello”)
- Syntax: `lambda [arg1,arg2,..]:[expression]`
  - start with the lambda keyword, then have:
  - a list of parameters,
  - a colon,
  - then a single statement.
- Can be assigned to a variable, to call it again later.
- File: lambda_func_1.py