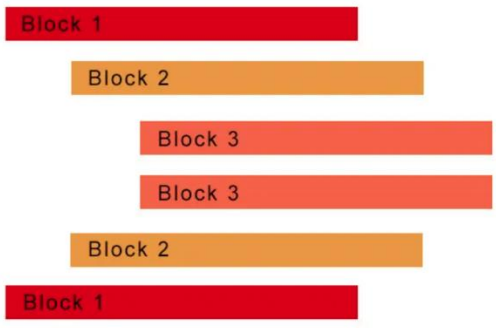
Intro to Python
Lab 2

1 Indentation and conditional statement: if - else - if

File to use: `if-else-basics.py`
File to use: `if-else-nested.py`

Python has a standard `if-else` and `if-elif-else` statement structures that match structures with identical functionality in other languages. It also can support nested `if-else` structures
Python supports the ternary operator, which is short cut for writing longer `if-else` and `if-elif-else` statement structures

It was mentioned in an earlier lab that Python uses whitespace and indentation to organize the code. Indentation is the leading whitespace (spaces or tabs) before any statement in Python. Python treats the statements with the same indentation level (statements with an equal number of whitespaces before them) as a single code block. A code block is a compound statement consisting of a group of basic statements that are executed as a single unit.
So whereas in languages like C, C++, etc. a code block is represented by curly braces { }, in Python a code block is a group of statements that have the same indentation level




```
def foo():
    print("Hi")
    if True:
        print("true")
    else:
        print("false")

print("Done")
```

1st level indentation
foo() method statements

2nd level indentation
if and else block code

Code without indentation
Belongs to the source file

Guidelines for indentation:

a) Any issues / errors with indentation will result in an IndentationError when executed. On Spyder, the integrated pyflakes static analysis tool will flag an error. Other IDEs have a similar way of flagging this error.
b) The first line of code in a Python source code file cannot be indented.
c) You should avoid mixing tabs and whitespaces to create an indentation. This is because certain text editors / IDEs behave differently with them and mixing them can cause the wrong indentation.
d) It is preferred to use whitespace than the tab character.
e) The best practice is to use 4 whitespaces for each indentation level. However, you can use any number of whitespaces you want as long you consistently adhere to it throughout your source code.

On Spyder, you can configure the indentation character types and length with:
Tools >>> Preferences >>> Editor >>> Source code >>> Indentation characters

2 Loops / iteration structure

File to use: `basic-for-loop.py`
File to use: `basic-while-loop.py`

Python has the same iteration / loop structures as other languages (`for, while`)
However, instead of using a loop counter variable to control the number of iterations of the for loop, Python uses the `range` built-in function to generate the loop counter values.

As is the case with `if-else` structure, the body of the loop is a code block with its own indentation level. This block itself can contain nested loops, if-else structures, etc with their own code blocks and so on, nested to any arbitrary depth.

The for loop is usually used when the number of iterations is known in advance. The `while` loop is usually used when the number of iterations is unknown, for e.g. when a loop body is executed based on specific user input. It is possible to obtain an infinite / endless loop when using `while`, so you must exercise caution
If this occurs, just type Ctrl-C in the IPython console to send a keyboard interrupt to terminate program execution.

2.1 Break, continue, pass

File to use: `break-continue-pass.py`

Python also has the keywords `break` and `continue` to fine tune the control of execution of a loop. These keywords have the same meaning and functionality as they do in other languages.
Python also includes a special keyword `pass` that is used as a placeholder for statements that are yet to be created in order to avoid compiler errors.

3 Functions

File to use: `function-basics.py`

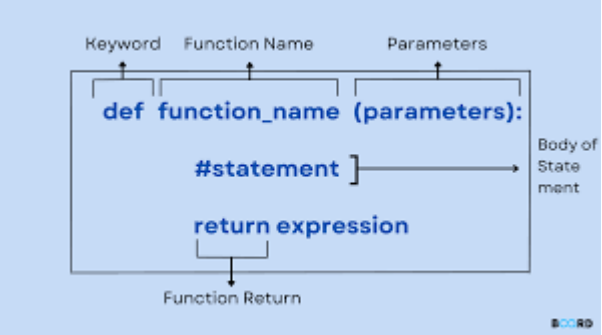
Functions in Python serve the same purpose as they do in other languages. They are blocks of code that perform a task that will need to be repeated multiple times in a larger program / script to avoid redundancy in our code.

The Python interpreter has a number of functions and types built into it that are always available: these are called built-in functions. Examples of this that we have already used are `print` and `range`.
https://www.programiz.com/python-programming/methods/built-in
These built-in functions are grouped within a special module called `builtins` (to be covered later)

To get documentation on a specific built-in function, you can type directly into the IPython console: `help(xxxx)`, for e.g. `help(print)` or `help(range)`

Here, we learn how to define our own custom functions for our own use.

a) A function has two main parts: a function definition and body.
b) A function definition starts with the `def` keyword followed by the name of the function.
c) The data that the function needs in order to accomplish its task is called parameters and this is provided in parenthesis after the function name.
d) The function body is a code block that must be at the same indentation level. This code block can contain all the other structures we have seen so far (conditional statements, loops, etc)
e) The function body can optionally include a `return` statement that specifies the value to be returned by the function
f) A function is called / invoked by specifying its name and the information it requires (arguments). Each argument value is passed to a matching parameter based on the exact sequence in which they appear.
g) The function must be defined before it is called.
h) Functions can optionally include a docstring at its start to provide documentation on it that can subsequently be accessed using `help()`



3.1 Default parameters and keyword arguments

File to use: `function-default-keyword.py`

Specifying default parameters allows a function to use those parameter values when the required arguments are not passed to it during function invocation.

When you call a function and pass an argument to the parameter that has a default value, the function will use that argument instead of the default value. However, if you don't pass the argument, the function will use the default value.
To use default parameters, you need to place parameters with the default values after other parameters. Otherwise, you'll get a syntax error.

3.2 Functions as first class citizens / objects

File to use: `functions-first-class.py`

As a highly object oriented language, functions are considered as first class citizens / objects (remember: in Python, **EVERYTHING IS AN OBJECT.)**
This means you can assign them to variables, store them in data structures (lists, tuples, etc), and **pass them as arguments to other functions, and even return them as values from other functions**

Functions that can accept other functions as arguments or return other functions as a result are called **higher-order** functions. They are used in the functional programming paradigm: which Python supports as well.
The classic use case for higher-order functions in Python is lambda functions used in conjunction with the various higher-order functions (map, filter) used to generate new lists (to be covered later)

Python is a multi-paradigm program that supports the four main paradigms: imperative, procedural, functional and object-oriented programming

3.3 Namespaces and variable scope in functions

File to use: `namespace-scope.py`

A namespace is a mapping from names (identifiers for variables, functions, etc) to objects. A scope is a region of the program where a particular identifier is directly accessible / visible (it defines the bounds of the namespace mapping). Each identifier has a scope, which is also referred to as its lifetime during execution. Many namespaces / scopes can exist at one time. Attempting to access a variable outside of its scope results in an error.

Python resolves names using the so-called LEGB rule, which is named after the scope names. The letters in LEGB stand for Local, Enclosing, Global, and Built-in

a) Local (or function) scope is the code block or body of any Python function or lambda expression. This Python scope contains the names that you define inside the function. These names will only be visible from the code of the function. It's created at function call, not at function definition, so you'll have as many different local scopes as function calls. Each call will result in a new local scope being created.

b) Enclosing (or nonlocal) scope is a special scope that only exists for nested functions. If the local scope is an inner or nested function, then the enclosing scope is the scope of the outer or enclosing function. This scope contains the names that you define in the enclosing function. The names in the enclosing scope are visible from the code of the inner and enclosing functions.

c) Global (or module) scope is the top-most scope in a Python program, script, or module. This scope contains all of the names that you define at the top level of a program or a module. Names in this scope are visible from everywhere in your code.

d) Built-in scope is a special scope that's created or loaded whenever you run a script or open an interactive session in IPython. This scope contains names such as keywords, built-in functions, exceptions, and other attributes that are built into Python. Names in this scope are also available from everywhere in your code. It's automatically loaded by Python when you run a program or script.

The LEGB rule is a kind of name lookup procedure, which determines the order in which Python looks up names. For example, if you reference a given name, then Python will look that name up sequentially in the local, enclosing, global, and built-in scope. If the name exists, then you'll get the first occurrence of it. Otherwise, you'll get an error

Global variables are useful for storing constant literals (such as the value PI) whose values are not expected to change during program execution. However, if the global variable is accessed at numerous times throughout the program in various functions (via the `global` keyword), this can result in unintentional changes to the variable at one location affecting code at another location. This makes the code more difficult to debug and maintain in the long run.

A better approach is to pass all the information that functions require to work as parameters, and then return the result to the code that calls them

3.4 Type hints and static type checking

File to use: `type-hints-check.py`

As explained before in a previous lab, Python uses dynamic typing. This provides a lot of flexibility in how we design our code, but can also be potential source of run time errors. This is particularly true when we accidentally pass arguments of the wrong type to a function that requires its matching parameters to be of a different type.

To resolve this issue, we can

a) Perform explicit type checking of the parameters received by the function before performing an operation that involves them, printing a message or throwing an exception if they are of the incorrect type
b) Use type hints and variable annotations for optional static typing. This allows us to leverage the best of both static and dynamic typing

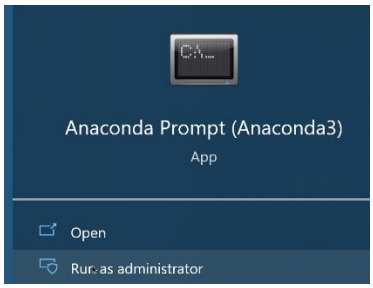
When you introduce type hints and variable annotations into your code, you can use a static type checker tool to check whether there is violation in the way that variables are being used based on their types. Python doesn't have an official static type checker tool. At the moment, the most popular third-party tool is Mypy. Since Mypy is a third-party package (we will talk about packages and package management in more detail in an upcoming lab), you need to install it:

For the Anaconda distribution, we can search for packages at:

https://anaconda.org/

Search for `mypy`. Select the `conda-forge/mypy` link.

You should see the instructions to install the package using Conda (the package manager for the Anaconda Python distribution)
Open an Anaconda prompt with Administrator privilege



and type the following to install:

```
conda install -c conda-forge mypy
```

Press Y to any prompts that appear.
In the Anaconda prompt, switch to the directory holding the current script.
Uncomment the statements that cause type violations and use the tool to analyze the script with:

```
mypy type-hints-check.py
```

Notice that while static type checking flags the errors, you can still run the script which will still result in the same run time error that we identified earlier.
In other words, it is not possible to disable or override the dynamic typing behaviour of Python. This is a core feature of the language.

This is why documenting a function properly (and in particular its parameter types and return types) using docstrings is extremely important to allow developers who intend to use it to invoke / call it correctly.