Toolkit for Finance

Week 4: More Control Flow and Working with Modules

Lecturer: Dr. Yiping LIN

Overview

- More Control Flow:

- Functions
- Comprehensions

- Working with Modules

- Creating a Module
    - From a script to a function
    - Docstrings and Comments
- Including test cases and examples using **name**
- A common file path error in Windows (Text, Unicode, and Bytes)
- Accessing functionality from other modules
- Create a Package

Notes: PyCharm Shortcut

- Only run selected block of code in Pycharm
    - select lines of code you wish to run
    - press option + shift + e key for Mac user
    - or press alt + shift + e key for Windows user
- Comment/uncomment a block of code
    - press command + / key for Mac user
    - press Ctrl + / key for Windows user
- Indent/Unindent a block of code
    - select lines of code you wish to move
    - press tab key to indent or press shift + tab key to unindent

1.0 More control flow

1. Functions
    - reusable pieces of code that are designed to perform specific tasks
2. Comprehensions
    - provide a concise way to construct lists, dictionaries, and sets containers.
    - can make your code easier to read and often execute faster than a corresponding loop.

1.1 Functions - Intro

Previously, we introduced the following example using loops:

```
import yfinance

start = '2020-01-01'
end = None

tickers = ["QAN.AX", "WES.AX"]
for tic in tickers:
    df = yfinance.download(tic, start, end)
    print(df)
    tic_base = tic.lower().split('.')[0]
    df.to_csv(f'{tic_base}_stk_prc.csv')
```

Every time you run this script, it will download prices for "QAN.AX" and "WES.AX", and the data will include all available prices starting from January 1, 2020, and be saved in the current working directory.
Imagine the following situation: You are working on some other project which requires stock price data. Specifically, you would like plot daily stock returns for Tesla's stock since its IPO in 2010.

- To do that, you create *another* script that uses Tesla's stock prices to compute returns. But how would you get Tesla's stock prices?

1.1 Functions - Intro

There are several approaches to consider.

1. Copy and paste the program above and change the values of the variables `start` and `tickers`. --> * Not a good idea, you will have different versions of the same program in different scripts.
2. Open the file with the above script, change the variables `start` and `tickers` and then execute the program. --> *Not a good approach, you will need to change the script* every* time you need different data.
3. There is a third approach -- write a function.
    - e.g., we could write a function that takes the variables `start`, `end`, and `ticker` as parameters. We would define this function in such a way that it could be accessed from other scripts.

Next, we will learn how to define and use functions.

- Functions are reusable pieces of code that are designed to perform specific tasks.
- A function allows us to specify its inputs (called parameters), enabling it to provide different results for different inputs.

1.1 Functions - Our first function

In the script above, the name of the CSV file was generated inside the loop for each value of the variable `tic`.
Consider the following statements:

```
tic = 'QAN.AX'                      #                        (1)
tic = tic.lower()                   # --> 'qan.ax'           (2)
tic_base = tic.split('.')[0]        # --> 'qan'              (3)
csv_name = f'{tic_base}_stk_prc.csv'  # --> 'qan_stk_prc.csv' (4)
print(csv_name)                     #                        (5)
```

The first statement creates a string instance and assigns it to the variable `tic`. Remember that Python uses namespaces to track down which objects are assigned to which variables.
All the variables defined above are stored in a namespace called *global*.

- This means that after the first statement, the *global* namespace will map the name `tic` to the `str` instance "QAN.AX". Whenever we modify the variable `tic`, the namespace will be updated.
- e.g., after `(2)`, the *global* namespace will be updated and the name `tic` will be mapped to the new string instance "qan.ax" (before it was "QAN.AX").

1.1 Functions - Our first function

The expression `tic.split('.')[0]` in statement `(3)` can be decomposed into:

```
tic  .split('.')[0]
  |  |     |   |  |
 (a)(b)   (c) (d)(e)
```

- `(a)` : Python will search the *global* namespace for the name "tic". At this point in the code, "tic" is mapped to the `str` instance "qan.ax". The easiest way to understand what happens next is to imagine that the variable `tic` is replaced by the `str` instance "qan.ax" stored in the computer memory.
- `(b)` : The "dot" tells Python that we want to access the attribute of the `str` instance "qan.ax".
- `(c)` : Python will search for the object assigned to the attribute `split` . This is a reference to the method `str.split` , which will "split" a string into a list using "." as a separator (in this case). Remember that `split` is *not* in the *global* namespace (we will come back to this point soon).
- `(d)` : At this point, `tic.split('.')` becomes the `list` instance `['qan','ax']` . We can access elements of lists using `[]` .
- `(e)` : Python will return the first element of the list `['qan', 'ax']` , in this case the `str` instance "qan".

1.1 Functions - Our first function

```
tic = 'QAN.AX'                          #                          (1)
tic = tic.lower()                       # --> 'qan.ax'             (2)
tic_base = tic.split('.')[0]            # --> 'qan'                (3)
csv_name = f'{tic_base}_stk_prc.csv'    # --> 'qan_stk_prc.csv'    (4)
print(csv_name)                         #                          (5)
```

Our goal is to create a function that will implement statements（2）to（4）every time we call it. We will do that in stages, experimenting with different versions of the function below:

```
def mk_csv_name(tic):
    tic = tic.lower()
    tic_base = tic.split('.')[0]
    csv_name = f'{tic_base}_stk_prc.csv'
    return csv_name
```

1.1 Function - Definitions

Functions are created using the `def` keyword in what is called a *function declaration* with the following format:

```
def func_name ( <parameter expression> ) :
    <function body>
```

In a function declaration, the keyword `def` is immediately followed by the function name (represented above by `func_name`). The function name must be followed by parentheses. Inside the parentheses, we can optionally define a set of inputs called *parameters*. The `<parameter expression>` above is meant to represent zero, one, or multiple parameters. In other words, we can define functions with multiple parameters or without any parameter at all. After the closing parenthesis, a colon ends the function declaration.

> *Note: Function declarations can include **annotations**, which we will not cover in this course.*

Function declarations tell Python that the user wants to define a function with a given name and a set of parameters.
The next step in creating a function is to tell Python what this function will do. These instructions are included in the *function body*, which is an indented code block following the declaration.

1.1 Function - Definitions

Functions will instruct Python to do something. All functions in Python will return an object. This means that when a function is called, it will do something *and* return some object.
Consider the following function:

```
def qan_tic():            # (1)
    tic = "QAN.AX"        # (2)
    print(tic)            # (3)
    return tic            # (4)
```

The function declaration

- `(1)` : consists of the keyword `def` , the name of the function (`qan_tic`), and *no* parameters.
  The function declaration is followed by an indented code block (the *function body*). The body contains the following instructions:
- `(2)` : Create a `str` instance "QAN.AX" and assign it to a variable called `tic`
- `(3)` : Print the value of the object assigned to `tic` (i.e., "QAN.AX") `tic`
- `(4)` : Return the object assigned to `tic` , in this case the `str` instance "QAN.AX".

1.1 Function - Definitions

Now that we understand what this code does, we should execute it and see what happens:

```
# Please execute the code below by pressing "Run"
def qan_tic():            # (1)
    tic = "QAN.AX"        # (2)
    print(tic)            # (3)
    return tic            # (4)
```

1.1 Function - Definitions

When you try to execute the code above, it may seem like nothing happened. This is not correct. What Python did was to *define* the function `qan_tic`. The reason nothing was printed is because we did not tell Python to execute the function `qan_tic`.
Important: A function defines a set of *instructions* that will be executed when the function is ***called***.

- Remember that we call functions using parentheses:

```
# Define a function called `qan_tic`
def qan_tic():            # (1)
    tic = "QAN.AX"        # (2)
    print(tic)            # (3)
    return tic            # (4)

# Call the function
qan_tic()                 # (5)
```

The reason we can see the printout "QAN.AX" is because of statement `(5)`. First, note that this statement is *not* part of the function body. The function body ends as soon as we leave the indented code block following the function declaration.
Instead, what statement `(5)` does is to ask Python to execute the instructions in the function body (which include `print(tic)`) *and* produce the object the function returns.

1.1 Function - Definitions

In the statement above, we did not assigned the output of `qan_tic` to any variable. Lets try that instead:

```
# Define a function called `qan_tic`
def qan_tic():            # (1)
    tic = "QAN.AX"        # (2)
    print(tic)            # (3)
    return tic            # (4)

# Call the function
res = qan_tic()           # (5b)
```

In this new version, statement `(5b)` will assign the string instance "QAN.AX" to the variable `res`. Summary so far:

- Statements `(1)` to `(4)` define a function called `qan_tic`. This function is a set of instructions (statements) that will be executed every time we call the function using `qan_tic()`.
- The `return` statement tells Python what this function will produce as a result. In this case, the output is the object assigned to the *function* variable `tic`.

1.1 Functions - The return statement

Return statements can be used inside functions (or methods) and consist of two parts, namely the keyword `return` and a Python expression:

```
return expression
```

The `expression` can be any valid Python expression. What the return statement does is to first evaluate the expression and then instruct Python to return the result.
All functions return an object but return statements are not required inside a function.

- If the user does not specify what the function should return, the function will return the `None` object.
- One way to think about this is to imagine that Python adds the statement `return None` to the body of any function we define.

1.1 Functions - The return statement

Once Python reaches a `return` statement inside a function, it will return the object created by `expression` and *exit* that function.

- In other words, no other statements inside the function body will be executed. To see that, lets revert the order of statements `(3)` and `(4)`:

```
# Define a function called `qan_tic`
def qan_tic():            # (1)
    tic = "QAN.AX"        # (2)
    return tic            # (4)     <-- Function will exit here!
    print(tic)            # (3)     <-- Will not be printed!

# Call the function
res = qan_tic()           # (5b)
```

As you can see, no output will be printed when you execute the code above. Even though the statement `print(tic)` is part of the function body, it will never be executed!

- Note: however, that the variable `res` still references to "QAN.AX", the value returned by the function.

1.1 Functions - The return statement

What happens if we remove statement `(4)` altogether?

```
def qan_tic():              # (1)
    tic = "QAN.AX"          # (2)
    print(tic)              # (3)

res = qan_tic()             # (5b)
```

In this case, the value of the function variable `tic` will be printed. However, the function will not return the value of `tic`! Since there are no return statements in the function body, this function will return `None`. This is what will be assigned to the variable `res` in statement `(5b)`:

```
def qan_tic():              # (1)
    tic = "QAN.AX"          # (2)
    print(tic)              # (3)

res = qan_tic()             # (5b)
print(res)                  # --> None
```

1.1 Functions - Scopes and namespaces

In the original code, statements `(1)` to `(4)` define a function called `qan_tic`. Like everything else in Python, this function is an object. The expression "define a function called `qan_tic`" means "create an instance of the `function` class, store it in the computer memory, and assign this object to a variable called `qan_tic`".
The code below will define the function but will not call it. Instead, we will print the object assigned to the *variable* `qan_tic`.

```
# Define a function called `qan_tic`
def qan_tic():            # (1)
    tic = "QAN.AX"        # (2)
    print(tic)            # (3)
    return tic            # (4)

print(qan_tic)
```

The code above will produce an output similar to:

```
<function qan_tic at 0x7f664f5ca5e0>
```

What this output mean is the following:

- The object being printed is a `function` instance.
- This instance is called `qan_tic`. The location (in the computer memory) where this function is stored is `0x7f664f5ca5e0`. This location will likely be different when you execute the code above.

1.1 Functions - Scopes and namespaces

Now, try the following:

```
print(qan_tic)

# Define a function called `qan_tic`
def qan_tic():            # (1)
    tic = "QAN.AX"        # (2)
    print(tic)            # (3)
    return tic            # (4)
```

This will produce the exception `NameError: name 'qan_tic' is not defined`. Why is that. The reason is that the variable `qan_tic` has not yet been defined.

1.1 Functions - Scopes and namespaces

What about the following code?

```
# Define a function called `qan_tic`
def qan_tic():            # (1)
    tic = "QAN.AX"        # (2)
    print(tic)            # (3)
    return tic            # (4)

print(tic)
```

Again, this will produce an exception. In this case, Python will complain that the variable `tic` has not yet been defined. However, we do define the variable `tic` in statement `(2)`!

- To understand why that happens we need to talk a bit more about ***namespaces and scopes***.

1.1 Functions - Scopes and namespaces

Imagine you walked into a actual library and asked the librarian for a book called "Python". Chances are the librarian would ask you to be more specific. This is because you could be referring to a book about the Python programming language, a book about snakes, or even a book about **Monty Python**! Conceivably, there could be multiple books called "Python" stored in different areas of the library. The librarian would need to know in which area to look to find you the correct book.
A *namespace* is a place where a variable is defined.

- Similar to how different books called "Python" are stored in different sections of the library, the same variable name can exist in different namespaces.
- The objects assigned to these variables can be very different as well, and have no relation with one another whatsoever.
- Since the same name can exist in different namespaces, Python needs to figure out where to look first.
    - ***The problem is that the way Python searches for a name depends the scope of that variable!***

1.1 Functions - Scopes and namespaces

The *scope* of a variable is the part of the program in which that variable has meaning. A variable defined inside function (like `tic` above) is local to that function. Variables defined in the main program environment are called *globals*. Consider the following code:

```
def qan_tic():
    tic = "QAN.AX"        # <-- local
    print(tic)
    return tic

tic = "WES.AX"            # <-- global
```

Depending on where we are in the program, the same name (`tic`) may refer to a local or global variable.

- The question is: if `tic` is both global and local, which value will be printed by `print(tic)`?

1.1 Functions - Scopes and namespaces

Remember that every time we define a variable, Python will store it in a namespace.

- Global variables are stored in the *global* namespace.
- Local variables are stored in the *local* namespace.
- Variables which are always available are defined in the *builtin* namespace.

Python will search for variable names in the order specified below. This is a sequential search, meaning Python will stop as soon as it finds a reference. If a variable cannot be found, a `NameError` exception will be raised.

- Local scope (**inside functions**):

1. If we are inside a function, the *local* namespace will be searched first.
2. Nested functions (which we have not discussed yet), create *enclosing* namespaces. If a variable cannot be found in the *local* namespace, Python will search this next.
3. After that, Python will search in the *global* namespace.
4. Finally, Python will search in the *builtins* namespace.

- Global scope (**outside functions**):

1. The global namespace is searched first
2. After that, Python will search the *builtins* namespace

The easiest way to understand this is to look at an example.

1.1 Functions - Scopes and namespaces

```
def qan_tic():              # (1)
    tic = "QAN.AX"          # (2)
    print(tic)              # (3)
    return tic              # (4)

tic = "WES.AX"              # (5)
print(tic)                  # (6)
qan_tic()                   # (7)
```

`(1)` : At this point, there are no user-defined variables in the *global* namespace. This statement is an instruction to create a function called `qan_tic` . We are in the global scope of the program so (after statements `(1)` to `(4)` are executed) the name of this function will be added to the *global* namespace.
`(2)` : We enter the local scope of the function. Variables defined in the function body will be added to the *local* namespace. As such, this statement will add `tic` to the local namespace (but *not* to the global namespace).
`(3)` : The name `print` is not part of the *local* namespace, so Python will search in the *global* namespace (which is "empty") and then in the *builtins* namespace. There it will find the function `print` . The interpreter will then search for `tic` in the *local* namespace. Statement `(2)` added `tic` to this namespace, so Python will find a reference to "QAN.AX". This is the object that will be printed.
`(4)` : The function will return "QAN.AX".

1.1 Functions - Scopes and namespaces

```
def qan_tic():              # (1)
    tic = "QAN.AX"          # (2)
    print(tic)              # (3)
    return tic              # (4)

tic = "WES.AX"              # (5)
print(tic)                  # (6)
qan_tic()                   # (7)
```

`(5)` : We are back in the global scope. Python will add the name `tic` to the *global* namespace.
`(6)` : Python will search for the name *print* in the *global* namespace (not there...) and then in the *builtins* namespace. It will search for `tic` in the *global* namespace, where it will find a reference to "WES.AX". This is what it will print.
`(7)` : Calling a function means "jumping back" into the local scope of that function. Although at this point the *global* namespace will contain a reference to "tic", this will be overwritten by the variable created in `(2)`. As a result, "QAN.AX" will be printed.
A valid question at this point: Does statement `(2)` overrides the value of `tic` in `(5)`?

- The answer is No. The reason is that `(2)` will not modify the *global* namespace.

1.1 Functions - Scopes and namespaces

What happens when we delete statement (2)?

```
def qan_tic():              # (1)
    print(tic)              # (3)
    return tic              # (4)

tic = "WES.AX"              # (5)
print(tic)                  # (6)
qan_tic()                   # (7)
```

- The string "WES.AX" will be printed twice.
    - Why? Inside the local scope of the function, Python will search in the *global* and *builtins* namespaces whenever a variable is not found in the *local* namespace.

1.1 Functions - Scopes and namespaces

Before moving on, why the following program will result in an exception?

```
def qan_tic():            # (1)
    print(tic)            # (3)
    return tic            # (4)

qan_tic()                 # (7)
tic = "WES.AX"            # (5)
print(tic)                # (6)
```

Extra points:

- Many other namespaces are created during the life of a program. This is because modules and classes generate their own namespaces.
- But Python does not search inside these namespaces automatically, unlike the ***local***, ***enclosing***, ***global***, **and *builtins* namespaces**.
- These other namespaces can only be searched using *qualified names*. We have already encountered qualified names before (like `str.upper`).

1.1 Functions - Shadowing

Variable *shadowing* refers to a situation when the same name is shared by variables at different scope levels. We encountered this situation before when we examined the following example:

```
def qan_tic():
    tic = "QAN.AX"          # <-- local       # (1)
    print(tic)                                # (2)
    return tic

tic = "WES.AX"              # <-- global
qan_tic()
```

In the code above, the local variable `tic` shadows the global variable `tic`. When the function is called, "QAN.AX" will be printed.
Although the variable `tic` is included in both the global and local namespaces, Python does not get confused because the local namespace takes priority over the global namespace.

1.1 Functions - Shadowing

Here is where things get interesting: If you swap statements `(1)` and `(2)` above, what do you see?:

```
def qan_tic():
    print(tic)                          # (2)
    tic = "QAN.AX"          # <-- local       # (1)
    return tic

tic = "WES.AX"              # <-- global
qan_tic()
```

The function call `qan_tic()` raises an `UnboundLocalError` with the message "local variable 'tic' referenced before assignment". How can this be the case?

- By the time the interpreter reaches `(2)`, the variable `tic` is defined in the *global* namespace. Logically, we would expect that "WES.AX" (not "QAN.AX") be printed.
- Python will raise exceptions whenever you reference a global variable that will become local later in the function (like the case above). The best way to think about this behavior is the following: Python is trying to warn that you might be making a mistake. Did you really mean to print the global variable `tic` before it became the local variable `tic`?

Tips: In general, try to avoid having variables with the same names at different scope levels. Reusing variable names at different scope levels can be a source of of confusion.

1.1 Functions - Parameters

So far, the function examples we discussed did not allow for any parameters. Functions that always execute the same statements are not particularly useful. Fortunately, we can pass objects around from the global scope of the code to the local function scope.
We specify function parameters as comma-separated names inside the function declaration.

- e.g., the function below will take a single parameter called `tic`:

```
def mk_csv_name(tic):                    # (1)
    tic = tic.lower()                    # (2)
    tic_base = tic.split('.')[0]         # (3)
    return f'{tic_base}_stk_prc.csv'     # (4)

name = mk_csv_name('QAN.AX')             # (5)
print(name)                              # (6)
```

1.1 Functions - Parameters

Parameters become local variables when the function is called. In other words, even though the expression `tic.lower()` in statement `(2)` occurs before any variable `tic` is defined, the code executes without a problem. In fact, you can call the function by explicitly defining the parameter as a variable:

```
def mk_csv_name(tic):                   # (1)
    tic = tic.lower()                   # (2)
    tic_base = tic.split('.')[0]        # (3)
    return f'{tic_base}_stk_prc.csv'    # (4)

name = mk_csv_name(tic='QAN.AX')        # (5b)
print(name)                             # (6)
```

In statement `(5b)`, we pass the parameter `tic='QAN.AX'`. Intuitively, the **local** variable `tic` is defined even before statement `(2)` is executed.

- Importantly, this `tic` variable is local: it does not exist outside the function.

1.1 Functions - Default parameters

Functions can have optional parameters as well. For example:

```
def mk_csv_name(tic, show=True):
    tic = tic.lower()
    tic_base = tic.split('.')[0]
    name = f'{tic_base}_stk_prc.csv'
    if show is True:
        print(name)
    return name

name = mk_csv_name('QAN.AX')
```

You can define Python function optional arguments by specifying the name of an argument followed by a default value when you declare a function.
In the example above, the parameter `show` is optional. This means that we can call the function without specifying a particular value for that parameter. If we do not, this parameter will take the default value specified in the function declaration (`show=True` in this case).

In-class Exercise 1

- Write a Python funcion to print the even numbers from a given list, and return the result as a list.
- Please test using this list [0,1,2,3,4,5,6,7,8,9,10,20,22,23,25,29,30,31]

```
### Sample Solution
test_lis = [0,1,2,3,4,5,6,7,8,9,10,20,22,23,25,29,30,31]

def is_even_num(lis):
    evennum = []
    for n in lis:
        if n % 2 == 0:
            evennum.append(n)
    return evennum

is_even_num(test_lis)
```

1.2 Comprehensions - Intro

Comprehensions provide a concise way to construct lists, dictionaries, and sets containers. Many *for loops* designed to populate containers can be replaced by comprehensions. Comprehensions will make your code easier to read and often execute faster than a corresponding loop.
Consider the following example:

```
# Create a list with all even integers from 0 to 1 million
evens = []
for x in range(1_000_000 + 1):
    if x % 2 == 0:
        evens.append(x)

print(evens[:10])
```

We can create the same list using a *list comprehension* :

```
# Create a list with all even integers from 0 to 1 million
evens = [x for x in range(1_000_000 + 1) if x % 2 == 0]
print(evens[:10])
```

Next, we will learn how to create and use comprehensions. We will also briefly discuss generator expressions. Let's start with dictionary comprehensions.

1.2 Comprehensions - Dictionary comprehensions

Suppose you want to create a dictionary from a list of (key, value) pairs. One way to do it would be:

```
pairs = [
  ('a', 1),
  ('b', 2),
  ('c', 3),
  ]
# Create an empty dictionary
dic = {}                        # (1)

# Iterate over each tuple in `pairs` and update the dictionary
for key, value in pairs:        # (2)
    dic.update({key:value})     # (3)

print(dic)
```

Our goal is to combine the statements `(1)`, `(2)`, and `(3)` into a single expression using a dictionary comprehension.

- To understand how a dict comprehension works, lets start with a thought experiment.

1.2 Comprehensions - Dictionary comprehensions

Suppose it was possible to combine `(2)` and `(3)` into a single line:

```
pairs = [
  ('a', 1),
  ('b', 2),
  ('c', 3),
  ]
# Create an empty dictionary
dic = {}                                          # (1)

# Iterate over each tuple in `pairs` and update the dictionary
dic.update({key:value}) for key, value in pairs       # (2 + 3)
```

This is very easy to read and understand:

- `(1):` "Create an empty dict"
- `(2 + 3):` "Update the dic for each key, value in `pairs`.

The problem is that statement `(2 + 3)` will not work in Python... The code above would raise a `SyntaxError` expression.

1.2 Comprehensions - Dictionary comprehensions

Since that does not work, lets try something else. What if we could combine `(1)` and the new `(2 + 3)` above into a single line?

```
pairs = [
  ('a', 1),
  ('b', 2),
  ('c', 3),
  ]
# This will NOT work
dic = {} dic.update({key:value}) for key, value in pairs
```

Of course, this will not work either. But take a closer look at the expression:

```
dic = {} dic.update({key:value}) for key, value in pairs
```

Although this is not a valid Python expression, it is more concise and easier to read than the original `(1)`, `(2)`, and `(3)`:

- Create a empty dict, populate it with the (key,value) pairs in `pairs` and assign it to the variable `dic`.

1.2 Comprehensions - Dictionary comprehensions

But we can change this expression slightly so that it *does* work:

```
pairs = [
  ('a', 1),
  ('b', 2),
  ('c', 3),
  ]
# This WILL work
dic = {key:value for key, value in pairs}
print(dic)
```

These types of statements are called *dict comprehensions*. In the case above, it creates a dictionary by iterating over the (key, value) pairs in `pairs`. It takes a while to get used to this syntax...

1.2 Comprehensions - Dictionary comprehensions

Here is another way to look at dict comprehensions: We can construct the same dictionary like this:

```
dic = {
  'a': 1,
  'b': 2,
  'c': 3,
  }
```

The dictionary above is created by "iterating" over several lines, where each line is `key: value`.
To create the same dictionary using a dict comprehension, replace these lines with a *for loop*:

```
pairs = [
  ('a', 1),
  ('b', 2),
  ('c', 3),
  ]
# This WILL work
dic = {key:value for key, value in pairs}
```

1.2 Comprehensions - List comprehensions

You can also construct lists using comprehensions.

- `newlist = [expression for item in iterable if condition == True]`

In the example above, we created a dictionary from a list of tuples (`pairs`).
Now, we will try the reverse -- We will start with a dictionary and then create a list of `(key, value)` tuples.
First, we will create the list without comprehensions:

```
# Create a dictionary
dic = {'a': 1, 'b': 2, 'c': 3}

# Create a list of (key, val) tuples called `pairs`
pairs = []
for key, value in dic.items():
    pairs.append((key,value))

print(pairs)
```

1.2 Comprehensions - List comprehensions

We can do the same with a list comprehension:

```
# Start with a dictionary
dic = {'a': 1, 'b': 2, 'c': 3}

# Create a list of (key, val) using list comprehensions:
pairs = [(key,value) for key,value in dic.items()]

print(pairs)
```

List and dict comprehensions are very common and we will use them often.

1.2 Comprehensions - Set comprehensions

You can also create sets using comprehensions. You can think of a set as a "dictionary without values". As such, you can create set comprehensions just like you create a dict comprehension, but without specifying the value:

```
# Start with a dictionary
dic = {'a': 1, 'b': 2, 'c': 3}

# Create a set comprehension with the keys from `dic`
keys = {key for key in dic}
print(f'The type of {keys} is {type(keys)}')
```

1.2 Comprehensions - Filtering

You can use comprehensions that only include elements which satisfy a certain condition. To do that, you include an `if` statement following the `for` loop inside the comprehension. For example:

```
# Create a list with all even integers from 0 to 1 million
evens = []
for x in range(1_000_000 + 1):
    if x % 2 == 0:
        evens.append(x)

print(evens[:10])
```

We can create the same list using a *list comprehension*:

```
# Create a list with all even integers from 0 to 1 million
evens = [x for x in range(1_000_000 + 1) if x % 2 == 0]
print(evens[:10])
```

1.2 Comprehensions - Is there a tuple comprehension?

Lets summarise what we have seen so far:

```
# Some data (list of tuples)
data = [
  ('a', 1),
  ('b', 2),
  ('c', 3),
  ]

# Create a dict comprehension
dic = {k:v for k, v in data}
print(f'`dic` is {dic}')
print(f'type(dic) is: {type(dic)}')

# Create a list comprehension
lst = [(k, v) for k, v in data]
print(f'`lst` is {lst}')
print(f'type(lst) is {type(lst)}')

# Create a set comprehension
st = {k for k, v in data}
print(f'`st` is {st}')
print(f'type(st) is {type(st)}')
```

1.2 Comprehensions - Is there a tuple comprehension?

It would be natural to assume that the following would create a tuple, but it doesn't!

```
# Some data (list of tuples)
data = [
  ('a', 1),
  ('b', 2),
  ('c', 3),
  ]

# Is this a tuple comprehension?
not_a_tup = (k for k, v in data)
print(f'The type of `(k for k, v in data)` is {type(not_a_tup)}')
```

Two important things to notice here:

1. There are no "tuple comprehensions" in Python. There are both practical and historical reasons for that. The main (historical) reason is the introduction of *generator expressions* by **PEP 289** back in 2002.
2. The `type` function above returned a type of object we have not seen before. What is this `generator` type?

1.2 Comprehensions - Generator expressions

We will not spend much time talking about generators and generator expressions in this course. Below you will find a *very* succinct discussion on generator expressions.
All the functions we have seen so far return a single object when called. For example:

```
# This function returns an `int` instance
def add(a,b):
    return a + b

# This function returns a `NoneType` instance
def useless():
    pass

# This function returns a `list` instance
def mk_a_list(a, b):
    return [a, b]
```

But you can also define functions that yield *a sequence of objects, one at at time*. These functions are called *generators*. Roughly speaking, generators return "loops" (technically speaking, they implement an iterator protocol). Generator expressions are a concise way of creating generators.

1.2 Comprehensions - Generator expressions

Let's look at an example. Consider the builtin `range(start, stop)` function, which returns a sequence of integers from `start` to `stop-1`:

```
# list comprehension --> `list` instance
lst = [i for i in range(10_000_000)]
```

The code above will create a list with 10 million observations. This list will be stored in the computer memory and then, the entire list will be assigned to the variable `lst`.
In contrast, the code below will create a *generator* and then assign this generator to a variable called `gen`.

```
# Generator expression --> generator
gen = (i for i in range(10_000_000))
```

The difference between `lst` and `gen` is the following:

- `lst` corresponds to an object (a `list`) with 10 million elements,
- while `gen` is an iterator that will return one element at a time.

1.2 Comprehensions - Generator expressions

Why is this useful?

There are many instances when we would like to read some type of data stream sequentially. For example, imagine we are interested in printing the contents of a huge text file.

- One way to do it would be to load the entire file into memory and then print each line in that file.
- Alternatively, we could load the first line in the file, print it, load the second line, print it, and so on... The advantage of the latter approach is that you do not have to load the entire contents of the file.

Similarly, the following statements will both sum all integers from 1 to 10M-1. The difference is that the first statement will create the entire list first, which will consume memory (and time) unnecessarily (extra spaces added for clarity).

```
bad_sum  = sum( [i for i in range(10_000_000)] )
good_sum = sum( (i for i in range(10_000_000)) )
```

In fact, you can ignore the double parenthesis and write simply `sum( i for i in range(10_000_000) )`

```
good_sum = sum( i for i in range(10_000_000) )
```

1.2 Comprehensions - Further reading

In this course, we will focus primarily on **list and dict comprehensions**. For the most part, we will not create complicated nested comprehensions or use complex filtering conditions. But comprehensions can be very flexible! The official Python Documentation is a good place to learn more about comprehensions:

- **List comprehensions**
- **Dict comprehensions**
- **Generator expressions (advanced)**

In-class Exercise 2

- Please use list comprehension to construct a list from the squares of each element in the following list
- Only if the value of the square is greater than 150.
    - lst = [2,3,10,14,20,21,25,13,15]

```
### Sample Solution
lst = [2,3,12,14,20,21,25,13,15]
new_lst = [x**2 for x in lst if x**2>150]
print(f'the new list with value of square greater than 150 is {new_lst}')
```

2.0 Working with Modules - Intro

In general, there are two main ways to interact with Python:

1. Interactive Python Shell.
    - There are multiple ways to access the Python interactive shell. In this course, we will use PyCharm's console.
2. Running code inside a module.

Typically, programmers use a combination of both to develop their programs.
The interactive shell is useful when you need to execute simple isolated statements or to check out documentation. You type in each command, one at a time, and inspect the results. Hence, we will use it to illustrate concepts inside Python.
However, the purpose of a programming language like Python is to perform tasks. Running code in an interpreter somewhat defeats that purpose. Thus, as a program develops, we will place code inside a *module*, which is a file containing a list of Python statements. We can run a module to execute the commands without necessarily needing human intervention.

The yf_example1.py module revisited

```
# Downloads Qantas share price beginning 1 January 2020
import yfinance                              # (1)
tic = "QAN.AX"                               # (2)
start = '2020-01-01'                         # (3)
end = None                                   # (4)
df = yfinance.download(tic, start, end)      # (5)
print(df)                                    # (6)
df.to_csv('qan_stk_prc.csv')                 # (7)
```

Although the script above works, it suffers from a number of drawbacks:

- To download data for another company, we need to open the script, change the value of the variable `tic`, and change the name of the output file, and then run the module again.
- This module will download the stock prices into the current working directory, the directory with the source code where Python was executed. It would be better locate the data files away from the source code.
- There is no documentation explaining what this module does. We actually have to read the entire file to understand its use.
- This module was saved directly under the PyCharm project folder. As we write more modules, the project folder will become increasingly cluttered.

2.1 Creating the yf_example1.py in PyCharm

Next, we will discuss how this script can be significantly improved.
The starting point is the `yf_example1.py` module we created previously. The goal is to find a better approach to download stock prices from Yahoo Finance.

- We will do that by creating new versions of the `yf_example1.py` module. To keep track of the differences, we will call these different versions `yf_example2.py`, `yf_example3.py`, etc...

To set the stage, please make sure that the file `yf_example1.py` is located directly under the `toolkit` project folder:

```
toolkit/
|___yf_example1.py
|...
```

2.1 Creating the yf_example1.py in PyCharm

If you deleted the file or saved it someplace else, please follow the steps below:

1. Open PyCharm and right-click on the *toolkit* project folder.
2. Choose *New >> Python File*
3. Name this file `yf_example1.py`
4. Open this module in PyCharm and copy and paste the following code

```
# Downloads Qantas share price beginning 1 January 2020
import yfinance                             # (1)
tic = "QAN.AX"                              # (2)
start = '2020-01-01'                        # (3)
end = None                                  # (4)
df = yfinance.download(tic, start, end)     # (5)
print(df)                                   # (6)
df.to_csv('qan_stk_prc.csv')                # (7)
```

This module will be our starting point. The next step is to create a function to download and save stock prices for any ticker and sample period.

2.1 From a script to a function

Consider the following statements (no need to copy any code into PyCharm for now):

```
import yfinance as yf

def yf_prc_to_csv(tic, pth, start=None, end=None):
    df = yf.download(tic, start=start, end=end)
    df.to_csv(pth)
```

The first thing you will notice is that we replaced `import yfinance` with `import yfinance as yf`. We also replaced `yfinance.download` with `yf.download`.

- To allow you to access the module with fewer keystrokes, you can choose to import the module under *aliased* identifier using `as` keyword.
- What the import statement does is to import the module `yfinance` and give it the alias `yf`. This means that the module (in the computer memory) will be assigned to a variable called `yf` and *not* `yfinance`.

The `yf_prc_to_csv` function above takes two required parameters and up to four parameters in total. The first parameter (`tic`) refers to the stock ticker. The second parameter (`pth`) is the location (in your computer) of the CSV file contained the downloaded prices. The `start` and `end` parameters are optional and can be used to select a specific sample period.

2.1 From a script to a function

Creating this function will give us more flexibility compared to the original script.

- For example, we can now download stock prices for any ticker `tic` and save the contents into a any file `pth` by calling `yf_prc_to_csv(tic, pth)`.
- Once this function is written, all that users need to know is *what* it does, not *how* it does it. This simplifies the amount of information you need to remember tremendously.

However, we have not yet provided any documentation to either the module or the function. It is very important that we properly document all our codes. Let's see an example next.

2.1 From a script to a function

```
""" yf_example2.py
Example of a function to download stock prices from Yahoo Finance.
"""
import yfinance as yf

def yf_prc_to_csv(tic, pth, start=None, end=None):
    """ Downloads stock prices from Yahoo Finance and saves the
    information in a CSV file

    Parameters
    ----------
    tic : str
        Ticker

    pth : str
        Location of the output CSV file

    start: str, optional
        Download start date string (YYYY-MM-DD)
        If None (the default), start is set to '1900-01-01'

    end: str, optional
        Download end date string (YYYY-MM-DD)
        If None (the default), end is set to the most current date available
    """
    df = yf.download(tic, start=start, end=end)
    df.to_csv(pth)
```

Note the triple-quoted strings which span multiple lines - *docstrings*, which will be discussed next.

2.1 Docstrings and comments

It is important that either you or other people are able to understand the use of your functions and modules without having to decipher any source code. The creator of Python puts it like this:

> ***"Code is more often read than written" -- Guido van Rossum***

There are two primary reasons for adding descriptive information to your code:

- provide high level ***documentation*** that tells others (and yourself) what a module or function does.
    - For example, we might want to tell others that `yf_prc_to_csv` downloads stock price data from Yahoo! Finance.
- write ***comments*** as reminders of how statements work or the logic behind a complex algorithm.
    - It's not necessarily intended for people that simply want to use your code.
    - Instead, they explain the inner working of your code to curious parties or to yourself as a reminder about important details.

2.1 Docstrings and comments

Python code is notated in two ways that match these purposes. The subtle difference between *documenting* and *commenting*:

- **Documentation** is used to describe the use and functionality of modules, functions, classes, and methods.
    - Docstrings are surrounded by triple quotes
    - It's automatically attached to the `__doc__` attribute of an object. This means that docstrings are easily available to functions like `help`. For example:

```
>>> def add(a,b):
...     """ Returns the sum of two numbers """
...     return a + b
>>> help(add)
Help on function add in module __main__:
add(a, b)
  Returns the sum of two numbers
```

- **Comments** are used to either explain parts of your *source code* which are not obvious, or for *tagging*. Think of tagging as notes to yourself. For example, should you wish to refactor code, you can leave a note such as (e.g., `# TODO: Refactor this section into a separate function`).
    - To comment a line of code -- Just place a hash symbol `#` before any statement.

2.1 Docstring styles

Over time, different docstring styles were developed, each with specific guidelines. The **NumPy Docstring Style** is both simple and widely used docstring guideline, which is why we will use it in this course.
You should change the default docstring format in PyCharm to take advantage of its code-completion functionality:

1. Click on *PyCharm >> Preferences*...
2. In the Preferences window, navigate to *Tools >> Python Integrated Tools*.
3. Under *Docstrings*, set the *Docstring Format* to "NumPy"

2.1 Docstring styles

Below, we summarise the parts of the guideline relevant for this course. We start with an example module, called `doc_example.py`, which is adapted from NumPy's **`example.py`**.

```
"""Docstring for the doc_example.py module.

Modules names should have short, all-lowercase names. The module name may have underscores if this improves
readability.

Every module should have a docstring at the very top of the file. The module's docstring may extend over multiple
lines. If your docstring does extend over multiple lines, the closing three quotation marks must be on a line by
itself, preferably preceded by a blank line.

This example was adapted from Numpy's `example.py <https://numpydoc.readthedocs.io/en/latest/example.html#example>`_

"""
```

2.1 Docstring styles

```
import os  # standard library imports first

# Do NOT import using *, e.g. from numpy import *
#
# Import the module using
#
#   import numpy
#
# instead or import individual functions as needed, e.g
#
#  from numpy import array, zeros
#
# If you prefer the use of abbreviated module names, we suggest the
# convention used by NumPy itself::

import numpy as np

# These abbreviated names are not to be used in docstrings; users must
# be able to paste and execute docstrings after importing only the
# numpy module itself, unabbreviated.
```

2.1 Docstring styles

```
def foo(var1, var2, long_var_name='hi'):
    """Summarize the function in one line.

    Several sentences providing an extended description.

    Parameters
    ----------
    var1 : array_like
        Array_like means all those objects -- lists, nested lists, etc. --
        that can be converted to an array.

    var2 : int
        Describe the type of the variable in more detail.

    long_var_name : {'hi', 'ho'}, optional
        Choices in brackets, default first when optional.

    Returns
    -------
    type
        Explanation of anonymous return value of type ``type``.

    Notes
    -----
    Notes about the implementation algorithm (if needed).

    """
    pass
```

2.1 Docstring styles

As this example illustrates, *any* module should start with the *module docstring*, with no empty line above it.

- The first line of the docstring should be either the module name, or a single-line description of the module.
- A comprehensive description of the module should be separated from this single-line descriptor by an empty line.
- The module docstring is followed by the import statements. You should separate the following groups by a single empty line: *Standard library imports first; Third party modules; then Modules you created*.
  For example:

```
import os             # ----------------
  import sys          # Standard library
  import pprint       # ----------------
                      # empty line
  import numpy as np  # --------------------------
  import pandas as pd # Third party modules you installed
  import yfinance     # --------------------------
                      # empty line
  import event_study  # Modules you created
```

2.1 Docstring styles

In general, docstring consists of a number of sections. Each section following the summary (including the extended summary, if any) should be separated by headings. Each heading should be underlined. The sections we will see in this course are:

- **Short summary:** A one-sentence summary that does not use variable names or the function name:

```
def add(a, b):
    """The sum of two numbers.

    """
```

- **Extended Summary:** A few sentences giving an extended description. This section should be used to clarify functionality, not to discuss implementation detail. For that, we will use the "Notes" section below.

2.1 Docstring styles

- **Parameters:** Description of the function arguments, keywords and their respective types.
  Enclose variable names in single backticks. The colon must be preceded by a space, or omitted if the type is absent. If it is not necessary to specify a keyword argument, use optional:

```
Parameters
----------
x : type
    Description of parameter `x`.

y
    Description of parameter `y` (with type not specified).

z : bool, optional
    Description of the optional parameter `z`. You can sepcify its
    default value here.
```

- **Returns** : Provides an explanation of the returned values and their types. Similar to the Parameters section, except the name of each return value is optional. The type of each return value is always required.

```
Returns
-------
int
    Description of anonymous integer return value.
```

- **Notes:** An optional section that provides additional information about the code, possibly including a discussion of the algorithm.

2.1 Best practices for comments and docstrings

- You should write complete docstrings *before* you write your code. Doing so does not mean you won't revise them later. However, writing docstrings first ensures that you will have thought about what that particular module, function, method, or class will do before you figure out *how* it will be done. It will give you perspective and help you identify flaws in your design.
- Try to avoid comments in your code unless necessary. Commenting should be used to explain sections of your code that are not immediately obvious. Comments should be succinct.
- Document your code as if you are writing them for someone else.
- Use "#" for comments and triple quotes for docstrings. Do not use triple quotes for comments.
- All modules, functions, classes, and methods should have docstrings. All function parameters should be described in the docstring.
- Consistency, consistency, consistency... Style guides exist for a reason so use them. Pick a docstring style (e.g., NumPy, Google) and stick with it.

2.1 From a script to a function

Back to the `yf_example2.py` example.
For now, simply create a new module in PyCharm called `yf_example2.py`.

- Copy and paste the previous `yf_example2.py` code (*with docstrings*) into this new module you created.
- Then, run this module by right clicking on the source code and selecting "Run...".

PyCharm will tell you that the code finished with exit code 0 (meaning, no errors). But nothing else will happen and no data will be downloaded.

- This is because all this module does is to define this function. It does not call the `yf_prc_to_csv` function anywhere.

When you open a ".py" in PyCharm and select "Run...", all statements inside that file will be sent to the Python interpreter. Python will then exit.

2.1 From a script to a function

A natural question here is "How can we use the function `yf_prc_to_csv` defined in the `yf_example2.py` file?
After you create the file `yf_example2.py`, your `toolkit` project folder will look like this:

```
toolkit/
|___yf_example1.py
|___yf_example2.py
|...
```

The picture above shows two *modules*, which are called `yf_example1` and `yf_example2`. The objects defined inside modules need to be imported before they can be accessed.
To see how that works, **start a new interactive Python session by opening the *PyCharm console*** and then type:

```
>>> yf_prc_to_csv('QAN.AX', 'qan_stk_prc.csv')
```

After you press enter, Python will raise the exception `NameError: name 'yf_prc_to_csv' is not defined`.

- The reason for that is the following: The name `yf_prc_to_csv` is not available in either the *global* or *builtins* namespaces.

2.1 From a script to a function

Before we can use `yf_prc_to_csv`, we need to tell Python where it is defined. We know that this function is defined in the `yf_example2` module. So perhaps the following will work:

```
>>> yf_example2.yf_prc_to_csv('QAN.AX', 'qan_stk_prc.csv')
```

Unfortunately, this will produce a similar `NameError`. Python will tell you that the name `yf_example2` (not `yf_prc_to_csv`) is not defined.

2.1 From a script to a function

Similarly to how we had to import the module `yfinance` to have access to the `yfinance.download` function, we need to import the module `yf_example2` to have access to the function `yf_example2.yf_prc_to_csv` function. Try the following:

```
>>> import yf_example2
>>> yf_example2.yf_prc_to_csv('QAN.AX', 'qan_stk_prc.csv')
```

What Python is doing is the following: In the first statement, `import yf_example2`, Python will execute the statements in the `yf_example2.py` file. Any object defined in that statement will be added to the `yf_example2` namespace. The module `yf_example2` will then be added to the *globals* namespace.

2.1 From a script to a function

It is crucial that you understand what is going on. The name `yf_prc_to_csv` is not part of the *global* namespace. Instead, it belongs to the `yf_example2` namespace, which is then nested into the *global* namespace. This means that the following will raise a `NameError` exception:

```
>>> import yf_example2
>>> yf_prc_to_csv('QAN.AX', 'qan_stk_prc.csv')
```

Once a module is imported, it will remain in the global namespace until we end that specific Python session.

As we will see shortly, the way to access the `yf_prc_to_csv` function from another module (another ".py" file) is very similar: We first import the module `yf_example2` and then call the function using `yf_example2.yf_prc_to_csv`.

Before we go any further, note that you can "reset" an interactive Python session in PyCharm

- by pressing the little curved arrow (`Rerun`) on the left of the console tab.
- When you reset the session, any name defined in the previous session (including modules imported) will be wiped from the *global* namespace.

2.2 Including test cases and examples using `__name__`

Previously, we called the function `yf_example2.yf_prc_to_csv` from inside an interactive Python session. The function executed without any issues. In practice, we should *test* every single function or module we define.
There are multiple ways of testing user-defined functions. Perhaps the easiest is to include a small example inside the module containing the function. Let's pretend we have not tested the function `yf_prc_to_csv` yet.
Open the `yf_example2.py` file in PyCharm and add the following statements to the end of the file:

```
# Example
tic = 'QAN.AX'
pth = 'qan_stk_prc.csv'
yf_prc_to_csv(tic, pth)
```

2.2 Including test cases and examples using `__name__`

This means that the `yf_example2` module becomes:

```
""" yf_example2.py

Example of a function to download stock prices from Yahoo Finance.
"""

import yfinance as yf

def yf_prc_to_csv(tic, pth, start=None, end=None):
    """ Docstrings
    """
    df = yf.download(tic, start=start, end=end)
    df.to_csv(pth)

# Example
tic = 'QAN.AX'
pth = 'qan_stk_prc.csv'
yf_prc_to_csv(tic, pth)
```

2.2 Including test cases and examples using `__name__`

We could then run the module and examine the output. The problem with this approach is that Python will execute the example above every time we import this module.

- For example, suppose you want to download stock prices for Tesla (the ticker is "TSLA"). We could create a new interactive Python session and then type:

```
>>> import yf_example2
>>> yf_example2.yf_prc_to_csv('TSLA', 'tsla_stk_prc.csv')
```

But these statements will download prices for *both* Qantas and Tesla.

- First, Python will download Qantas prices when the `import yf_example2` statement is executed.
- It will then download Tesla's prices in the second statement.

2.2 Including test cases and examples using `__name__`

Fortunately, we can make our sample code execute *only* when we run the module directly. In other words, the example inside `yf_example2.py` will be executed only when we open the file in PyCharm and select "Run...".
This improved approach uses a special variable called `__name__`.

- This variable is automatically created by Python and will be assigned a different value depending on whether you execute or import the module.
- Importantly, you do not create this variable yourself (in fact, you should not create variables that start with a double underscore).

The `__name__` variable behaves as follows:

- When you execute a module, the value of `__name__` will be the string `"__main__"` *only* inside the executed module.
- When you import a module, the value of `__name__` will be the name of the module.

2.2 Including test cases and examples using `__name__`

Lets go back to the `yf_example2.py` module and look at the value of `__name__` under the two different scenarios we considered:

- When we use "Run..." to execute this module, the value of `__name__` will be `"__main__"`.
- When you import the `yf_example2.py` module using an import statement, the value of `__name__` will be the string `"yf_example2"`.

This means that we should expect a statement like `print(f"The value of __name__ is: '{__name__}'")` to produce a different output depending on whether we execute or run a module.

Now, go back to the `yf_example2.py` module and include the following at the end of the file:

```
# REMEMBER TO DELETE THIS!!!
# This will print the value of __name__
print(f"The value of __name__ is: '{__name__}'")
```

Then execute the module using "Run..." to get the following output:

```
The value of __name__ is: '__main__'
```

2.2 Including test cases and examples using `__name__`

Now, open a new interactive shell (or reset the previous one) and type:

```
>>> import yf_example2
```

Which will give you:

```
>>> import yf_example2
The value of __name__ is: 'yf_example2'
```

Note: You must reset the previous interactive shell for this to work! This is because Python will not* import the same module twice.

2.2 Including test cases and examples using `__name__`

Why is this useful? Remember that we would like to include examples inside modules to test functions (or even illustrate their use). At the same time, we don't want this example to be executed when we import the module.
To see that, first delete the previous print statement and then consider the following new version of `yf_example2.py`:

```
""" yf_example2.py

Example of a function to download stock prices from Yahoo Finance.
"""
import yfinance as yf

def yf_prc_to_csv(tic, pth, start=None, end=None):
    """ Docstrings

    """
    df = yf.download(tic, start=start, end=end)
    df.to_csv(pth)

# Example
if __name__ == "__main__":
    tic = 'QAN.AX'
    pth = 'qan_stk_prc.csv'
    yf_prc_to_csv(tic, pth)
```

2.2 Including test cases and examples using `__name__`

- When we execute this module (using "Run..."), the value of `__name__` will be the string `"__main__"`. In this case, the condition in the `if` statement at the end of the module is satisfied and the example code will be executed.
- However, if we import this module, the value of `__name__` will *not* be `"__main__"`, and the statements under `if __name__ == "__main__"` will not be executed.

To see that:

1. Run the module `yf_example2.py` (using "Run...")
2. Note the new `'qan_stk_prc.csv'` file under the `toolkit` folder
3. Delete the CSV file (careful not to delete the modules!)
4. Open the shell and type `import yf_example2`
5. Note that the function will not be executed
6. You can go back and run the module again

We will use this little trick extensively... For brevity, we will refer to these statements as the "statements under `if __name__ ...`". We will only use the variable `__name__` in `if __name__ == "__main__"` code blocks so the meaning should always be clear.

2.3 Text, Unicode, and Bytes

A quick recap so far:

1. We started with the `yf_example1.py` module, which included a simple script to download Qantas stock prices.
2. The new module `yf_example2.py` defines a function called `yf_prc_to_csv` to download stock prices for any ticker.
3. We added an example inside the `if __name__ ...` code block, which downloaded the `qan_stk_prc.csv` file.

Note that the `qan_stk_prc.csv` file was created directly under the `toolkit` project folder.

```
toolkit/
|__ yf_example1.py      <-- module
|__ yf_example2.py      <-- module
|__ qan_stk_prc.csv     <-- data file
```

2.3 Text, Unicode, and Bytes

We would like to have more control over where the information downloaded from Yahoo! Finance is saved on our computer (file path).

- Later in the course, we will also look at ways to read and write text files using Python.
- For now, let's discuss a common error when specifing file path (related to Text, Unicode, and Bytes):

```
>>> DIR = 'C:\User\Someone'
  File "<input>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in position 2-3: truncated \UXXXXXXXX escape
```

The reason for this error is because Python thinks the "\U" inside the string is an instruction to look for a Unicode code point (instead of a representation of "\User"). Because it cannot find a corresponding code point, it will raise an exception.

- Read *Text, Unicode, and Bytes* section first, and we can discuss this in tutorial.

To prevent this behaviour, you can either escape the backslash character itself through a double backslash `\\`, or you can tell Python to interpret the characters inside the quote as a *raw string* by prefixing the opening quote with "r".
For example,

```
>>> DIR1 = 'C:\\User\\Someone'
>>> DIR2 = r'C:\User\Someone'
```

2.4 Accessing functionality from other modules

The `yf_example2.py` module includes a function to download stock prices from Yahoo Finance. We already know that running this module in PyCharm will also execute all the statements under the `if __name__ == "__main__"` expression:

```
if __name__ == "__main__":
    tic = 'QAN.AX'
    pth = 'qan_stk_prc.csv'
    yf_prc_to_csv(tic, pth)
```

As it is, this will save the "qan_stk_prc.csv" file will under the current working directory, which in this case is the `toolkit` project folder.

```
toolkit/                <-- PyCharm project folder
|   ...
|   qan_stk_prc.csv     <-- Original location of the CSV file
|   ...
```

2.4 Accessing functionality from other modules

A better approach is to create a new folder, called `data/`, which will hold all the files we download. For simplicity, we will create this folder inside the `toolkit` project:

```
toolkit/                <-- PyCharm project folder
|   ...
|___data/               <-- New folder
|     qan_stk_prc.csv   <-- New location of the CSV file
```

Create this new `data` folder yourself. You can do so by right clicking on the `toolkit` project and then choosing *New >> Directory*. Make sure you call this directory "data" (in lower case letters). You can delete the `qan_stk_prc.csv` file at this point.

2.4 Accessing functionality from other modules

After creating this folder, we are ready to make the necessary modifications to the `yf_example2.py` module. Open this module in PyCharm (but do not run it yet). At the end of the file, we have:

```
if __name__ == "__main__":
    tic = 'QAN.AX'
    pth = 'qan_stk_prc.csv'
    yf_prc_to_csv(tic, pth)
```

These statements allowed us to download Qantas stock prices by running the `yf_example2.py` module in PyCharm. We can modify the definition of the variable `pth` so it points to a file inside the `data` folder. But to do that, we need to know where this folder is located in the computer.

- For example, if the `data` folder is located at `/home/fintec/pycharm/toolkit/data/`:

```
if __name__ == "__main__":
    tic = 'QAN.AX'
    datadir = '/home/fintec/pycharm/toolkit/data'
    pth = f'{datadir}/qan_stk_prc.csv'
    yf_prc_to_csv(tic, pth)
```

This is (almost surely) *not* where the `data` folder is located in your computer. If you use Windows, the correct location in your computer will look very different and will likely start with "C:\Users".

2.4 Accessing functionality from other modules

To find the location of the `data` folder, right click on the folder in PyCharm and select *Copy Path...* Then, paste the location of this folder in the `datadir` statement above.
IMPORTANT: If you are a Windows user, please note directories starting with `C:\Users` are very common. Strings literals containing escape sequences like `\uXXXX` will be interpreted as references to Unicode code points. As discussed earlier, you can prevent this by prefixing the string with "r" (to create raw strings) or including double backslashes:

```
# Using double backslashes
datadir = 'C:\\Users\\some\\other\\directory\\toolkit\\data'
pth = f'{datadir}\\qan_stk_prc.csv'

# using raw strings
datadir = r'C:\Users\some\other\directory\tookit\data'
pth = fr'{datadir}\qan_stk_prc.csv'
```

PyCharm will sometimes convert backslashes into forward slashes (in Windows systems). You should **not** rely of these types of conversions. If you are using Windows, you should develop the habit of escaping backslashes yourself (or use raw strings).
After you modify the module to include a path to your data folder, run the code again to make sure it worked.

2.4 Accessing functionality from other modules

This seems to work fine! A valid question at this point would be "Why not add the `datadir` variable to the body of the function directly?"
In general, it is a *very* bad idea to include paths like the one above inside functions and modules because:

- The code will not be *portable* -- Windows uses backslashes to separate folders and files, while non-Windows system use forward slashes. Also, the location of the files and folders will likely be different.
- It makes it very difficult to keep track and update the location of the `data` folder. If we define a variable similar to `datadir` above every time we need to save a file in that folder, we will have to update all these variables every time we have to change the location of the `data` folder.

2.4 Accessing functionality from other modules

To solve the first problem, we will use a method called `os.path.join`, which is part of the `os` module in Python's standard library.

- Modules in the Python standard library do not need to be installed (they are part of Python), but they *do* need to be imported.
- The `join` function we are trying to use belongs to the `os.path` module, which is in turn part of the `os` module (confusing?).
- To make sense of what is going on, remember that we use qualified names (names separated by a "dot") to tell Python where to look for that name.

2.4 Accessing functionality from other modules

As the name suggest, the `os.path.join` is used to combine different parts of a path. The advantage is that it will automatically take into account differences in the way operating systems separate folders.

- Suppose the location of the `data` folder is specified in a variable called `datadir`. Then, the value of `pth` below will depend on the operating system:

```
import os
pth = os.path.join(datadir, 'qan_stk_prc.csv')
```

This code will produce different values of `pth` depending on the operating system:

- Windows, same as `pth = f'{datadir}\\qan_stk_prc.csv'`
- Mac/Linux/Unix, same as `pth = f'{datadir}/qan_stk_prc.csv'`

2.4 Accessing functionality from other modules

In principle, this means modifying the module `yf_example2.py` as follows (no need to modify anything just yet):

```
if __name__ == "__main__":
    import os
    tic = 'QAN.AX'
    datadir = '/home/fintec/pycharm/toolkit/data'
    pth = os.path.join(datadir, 'qan_stk_prc.csv')
    yf_prc_to_csv(tic, pth)
```

Note that we are *not* including the `import os` statement at the beginning of the module. It is good practice to include all import statements at the top of the file. This is true, but we have a valid reason (actually, more of an excuse) to do that -- This is just a temporary fix.

2.4 Accessing functionality from other modules

Including the `os.path.join` method is an improvement, but it is still a bad idea to define the location of the `data` folder inside this module.

- Over the next few weeks, we will write many modules, which will save different files. We want all these files to be saved under the same `data` folder.
- In other words, we want the variable `datadir` to be available to *all* modules inside the `toolkit` project folder.

It is very common, and in fact very good practice, to define variables like this one in separate module. This module, which we will call `toolkit_config.py`, will include the definition of all variables that should be accessible to *any* module in this project.
Create a new module called `toolkit_config.py` directly under the `toolkit/` folder with the following content:

```
""" toolkit_config.py

Project configuration file
"""
import os

PRJDIR = '???'
DATADIR = os.path.join(PRJDIR, 'data')
```

2.4 Accessing functionality from other modules

This module defines two constants, `PRJDIR` and `DATADIR`. By convention, constants are expressed using upper case characters. These will hold the path to the `toolkit` project folder and its `data` sub-folder. You need to replace the "???" with a *string* containing the location of the `toolkit` folder *in your computer*.

- To do that, first open PyCharm, right click on the `toolkit` project, and choose *Copy >> Copy Path*. If you are given the option, choose *Absolute Path*. Then paste the path inside the string literal.

Suppose the location of your toolkit project folder is `/home/fintec/pycharm/toolkit`. In that case, your `PRJDIR` variable should be defined as:

```
PRJDIR = '/home/fintec/pycharm/toolkit'
```

If you are a Windows user, remember to either use double backslashes or to prefix the string with "r" (to create raw strings):

```
# Using double backslashes
PRJDIR = 'C:\\Users\\some\\other\\directory'

# using raw strings
PRJDIR = r'C:\Users\some\other\directory'
```

2.4 Accessing functionality from other modules

Important: You will use the `toolkit_config.py` module for the rest of this course.
Once the `toolkit_config.py` module is done, you can modify the `yf_example2.py` module so that it looks like this:

```
if __name__ == "__main__":
    import os
    import toolkit_config as cfg
    tic = 'QAN.AX'
    pth = os.path.join(cfg.DATADIR, 'qan_stk_prc.csv')
    yf_prc_to_csv(tic, pth)
```

If you run this module, the data will be downloaded to a CSV file called `qan_stk_prc.csv` inside your `data` folder.

2.5 The lectures package

Going forward, it is crucial that you implement all the Python code discussed in the lectures yourself. To help you with that, we will provide you with several modules or "companion code". The goal is to give you a "scaffold" to help you "try out" the code yourself. We will include these modules in a separate package, which we will call `lectures`.
This means we are in a good position to discuss the concept of a Python package in more detail. Remember that a package is a folder containing Python modules (possibly including other packages) *and* a module called `__init__.py`. When you import a package, the contents of the file `__init__.py` will be executed. However, other modules inside the package will not be automatically imported.
To see how that works, lets create a new package.

- Open PyCharm and right click on the `toolkit` project folder.
- Then select *New >> Python Package* and give the name `lectures`.
    - Note that you should choose *New >> Python Package* instead of *New >> Directory*.
    - The difference is that choosing *New >> Python Package* will automatically create a module called `__init__.py` (which Python uses to differentiate packages from simple folders). This initial `__init__.py` file will be empty.

2.5 The lectures package

After you create the `lectures` package, create a new module called `mod_inside_lec.py`. The location of this module is below:

```
toolkit/                <-- PyCharm project folder
|   ...
|___lectures/           <-- New package
|     mod_inside_lec.py <-- New module
```

Then, open this module and include the following content:

```
""" mod_inside_lec.py

A module inside the `lectures` package
"""

print("You can see me!")
```

Now, run this module (right-click and select "Run...") and you should see the output as expected.

2.5 The lectures package

The next step is to create another module inside the same package, called `another_mod_inside_lec.py`:

```
toolkit/
|   ...
|___lectures/
|     mod_inside_lec.py
|     another_mod_inside_lec.py   <-- New module
```

Open the new module and copy/paste the following:

```
""" another_mod_inside_lec.py

Another module inside the `lectures` package
"""
import mod_inside_lec
```

If you run this module in PyCharm (right click and select "Run..."), it will give you the "You can see me!" expression as an output.

2.5 The lectures package

But what is really happening when we execute this module? How does Python know where to find the `mod_inside_lec.py` module when evaluating the `import mod_inside_lec` above? In general, how does Python find modules?
Intuitively, Python will look for a module called `mod_inside_lec` using the following procedure:

- Python looks for a file called `mod_inside_lec.py` in a list of locations called the "Python system path". Think of this system path as a list of folders that will be searched by Python sequentially.
- If this module is found, its contents will be imported and the search will stop.
- If this module is not found, Python will raise an exception and the program will stop.

The list of locations included in the system path can be changed by the user. This involves calling the `sys` module and appending a new location to the `sys.path` list. Like `os`, `sys` is also part of the Python Standard Library.

- In this course, we will not be modifying the `sys.path` ourselves. Instead, we will rely on PyCharm.

2.5 The lectures package

Lets go back to the example above.
We mentioned that opening the module `another_mod_inside_lec.py`, right-clicking on the source code, and choosing "Run..." will instruct PyCharm to tell Python to execute the statements inside `another_mod_inside_lec.py`.
This is true, but we did not mention that PyCharm will manipulate the system path *before* executing the module.

- **By default, PyCharm will add the location of the module (i.e.,the folder `lectures` in our example) you are running to Python's system path.**

2.5 The lectures package

Intuitively, this is what will happen when you run the `another_mod_inside_lec.py` in PyCharm:

- PyCharm will figure out the location to the module you want to run, in this case, the folder `lectures`.
- PyCharm will tell Python to include this folder in the list of locations it uses to search for modules and packages.
- PyCharm will call the Python program on the module `another_mod_inside_lec.py`.
- Python (not PyCharm) will then execute this module:
    - When evaluating `import mod_inside_lec`, Python will look for a module called `mod_inside_lec`
    - Since `lectures` is among the folders in Python's search path, and there is a file called `mod_inside_lec.py` inside `lectures`, this module will be imported.
    - Importing the `mod_inside_lec` module will execute the `print("You can see me")` statement

2.5 The lectures package

All good and convenient...
But try the following next: Instead of "right-click Run..." the `another_mod_inside_lec.py` module, open a new Python interactive shell (just click on the *Python Console* tab). Then, type the following command:

```
>>> import another_mod_inside_lec
```

Unfortunately, this will *not* work and you will get the following exception:

```
ModuleNotFoundError : No module named 'another_mod_inside_lec'
```

Why is that?

- **Here when you open the Python Console in PyCharm, it will add *only the location of the toolkit project folder* to the list of directories Python will look for modules and packages.**

Intuitively, this is what happens when you open a shell and issue the `import another_mod_inside_lec` command:

- When you start the shell, PyCharm will add the location of `toolkit` to Python's system path.
- After you type the `import another_mod_inside_lec` command, Python will look for a file called `another_mod_inside_lec.py` inside the `toolkit` folder. No such file exists inside `toolkit` -- it's inside `lectures` -- so Python raises an exception.

This is very important. Python will look for both *modules and packages* inside each folder in the search path. You can access any module inside `toolkit` (like `yf_example2`) directly.

2.5 The lectures package

To access modules inside packages, you need to tell Python to look for a module inside that particular package. That is, to access *modules inside packages* you need to provide the package name first, and then refer to the module inside this package.
Your PyCharm project structure looks like this:

```
toolkit/                       <-- PyCharm project folder
|   ...
|___lectures/                  <-- the `lectures` package
|     mod_inside_lec.py        <-- the `lectures.mod_inside_lec` module
|     another_mod_inside_lec.py <-- the `lectures.another_mod_inside_lec` module
```

When you open `another_mod_inside_lec.py` in PyCharm and choose "Run...", PyCharm will include the `lectures` folder in the search path.

- **However, when you use the Python Console, the `toolkit` folder is included in the path, but the sub-folder `toolkit/lectures` is not.**
- **To get access to the `another_mod_inside_lec`, we need to tell Python that this module is inside the package `lectures`.**

2.5 The lectures package

Go back to the **Console** and try the following (note that we are importing `mod_inside_lec` **NOT** `another_mod_inside_lec`):

```
from lectures import mod_inside_lec
```

This will work because we are telling Python to:

- Look for a package called `lectures`. Since PyCharm adds the `toolkit` folder to the system path, Python will find this package.
- Look for a module inside `lectures` called `mod_inside_lec`. Since this module exists, Python will import it.

Now, try to import the other module using a similar statement:

```
from lectures import another_mod_inside_lec
```

This will fail, but for a different reason than before!

```
ModuleNotFoundError: No module named 'mod_inside_lec'
```

Note that the module Python could not find was `mod_inside_lec` and not `another_mod_inside_lec`.

This actually makes perfect sense because Python will:

- Look for a package called `lectures`. Since PyCharm adds the `toolkit` folder to the system path, Python will find this package.
- Look for a module inside `lectures` called `another_mod_inside_lec`. Since this module exists, Python will import it.
- Inside `another_mod_inside_lec` there is another import command, `import mod_inside_lec`.
- Python will try to execute this command as follows:
    - Look for a module called `mod_inside_lec`. There is no such module inside `toolkit`!
    - Python will raise an exception and stop the code.

2.5 The lectures package

The problem is that we are trying to import a module called `mod_inside_lec` without telling Python this module is inside a package!

- So, open the `another_mod_inside_lec.py` module in PyCharm and change it to:

```
""" another_mod_inside_lec.py

Another module inside the `lectures` package
"""

# OLD: import mod_inside_lec
from lectures import mod_inside_lec
```

Then, go back to the Console and try it again:

```
from lectures import another_mod_inside_lec
```

You should get no errors from this import statement.

The bottom line: If a module is inside a package, you should include the name of the package when importing the module using either `from package import module` or `import package.module`.