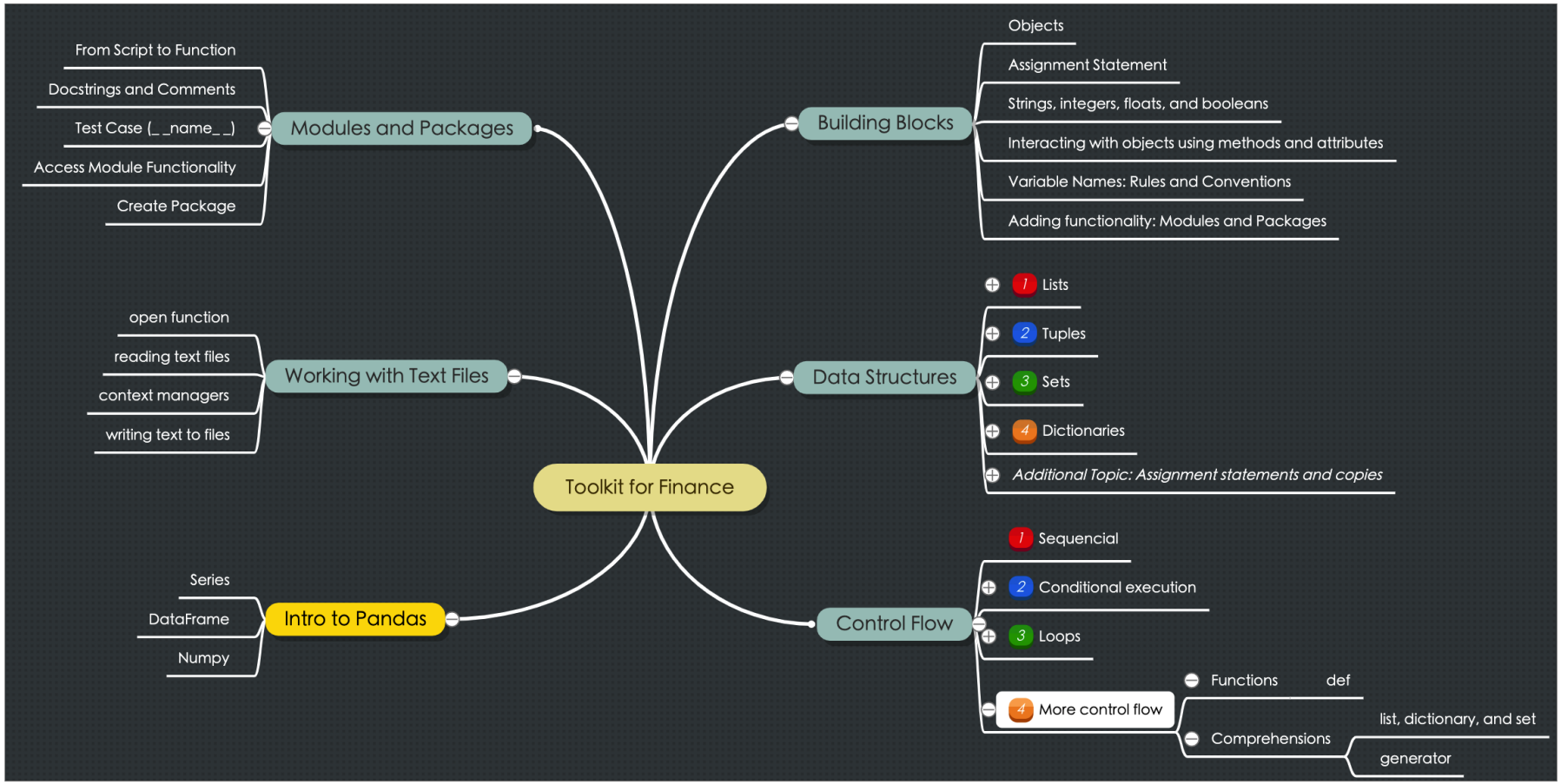
Toolkit for Finance

Week 2: Financial Analysis with Python: the Building Blocks

Lecturer: Dr. Yiping LIN

First Half Mindmap



Points to be Aware of

- Make sure your Pycharm and Python are installed properly (ready to code during the class as we go);
- We will not go through every single point in details in class;
- There are many concepts and jargons introduced in the first half of the course, and some may take a while to digest;
- There will be a few in-class exercise questions which you will be given some time to think and attempt.
- This slides will be made available to you if needed but essentially it's the same as the notes released on Ed.

Recap

1. Financial Analysis with Python: Downloading stock prices

- Introducing Python
    - A Python program or code is a collection of instructions called statements.
    - Programs are saved in files called modules. These files must have a `.py` extension.
    - A package is a folder containing modules and/or other packages.
    - A library is a collection of modules and/or package designed to implement some functionality.
- First task: Downloading Qantas stock price from Yahoo! Finance
- PyCharm

2. Python: The building blocks I

1. Objects
    - *Classes* are factories that create *instance* objects of a certain *type*
    - The *class* defines which interactions are possible
2. Assignment Statements
    - `variable = expression`

Overview

1. Python: The building blocks II
    - 1.The Building Blocks: Strings, integers, floats, and booleans
    - 2.Interacting with objects using methods and attributes
    - 3.Variable Names: Rules and conventions
    - 4.Adding functionality: Modules and packages
2. Data Strucutres
    - 1.List
    - 2.Tuple
    - 3.Set
    - 4.Dictionary

1.1 The Building Blocks: Strings, integers, floats, and booleans

For the sake of exposition, we will classify objects in Python into categories: Primitive and compound (or non-primitive).

- Primitive objects are the core building blocks of the language including textual data (strings), numerical data (integers and floating points), and True/False data (Booleans).
- We will call all other types "compound" - because they assemble data from the primitive types (or other compound types) into larger data types.
    - e.g., a street address may consist of an integer building number and a string street name. Hence, the street address is a compound type of integers and strings.
- Primitive types behave in a straight forward intuitive manner. Compound types on the other hand can behave differently and often lead to coding mistakes for new programmers. We will start with primitive types.

> *IMPORTANT: The distinction between primitive and compound is purely for pedagogical purpose and Python itself does not differentiate this.*

1.1 Strings: A type for textual data

Let's return to our code to download Qantas price data:

```
# 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)
```

The initial `import` statement (which we will discuss later) is followed by an assignment:

```
tic = "QAN.AX"        # (2)
```

Intuitively, this statement is an instruction to assign Qantas ticker to the variable `tic`. The ticker is textual information.

- Textual data are called *strings* in nearly every programming language because text is literally a series, or string, of characters.

1.1 Strings: A type for textual data

You may be tempted to try the following code to assign the Qantas ticker to the `tic` variable:

```
# Try it!
tic = QAN.AX  #`NameError`: name 'QAN' is not defined
```

In Python, you surround text with quotes to create a string object:

```
tic = 'QAN.AX'
```

If you compare this to the original code, you will notice that we used single quotes in the line above, but double quotes in the original code:

```
# Original statement:
tic = "QAN.AX"
```

1.1 Strings: A type for textual data

It is important to note that the ending quotation symbol must match the beginning one.

- For example, if you use double quotes to start a string, you cannot use a single quote to end it.

```
# Try it!
x = "<-- double quote but single quote -->'
```

The error message will read something like this:

```
  SyntaxError: EOL while scanning string literal
```

The `SyntaxError` exception tells you that you violated the Python syntax. Specifically, the interpreter reached the end of the line (EOL) and did not find a quote matching the initial double quote. In other words, it did not find the end of the string.

1.1 Strings: A type for textual data

You can define a string in Python with single or double quotes. You can even use triple single or triple double quotes. Triple quotes have a special meaning in Python. For now, just note that the following are all valid ways to create strings:

```
string1 = "Monty Python's Flying Circus"
string2 = 'I like the "Dead Parrot" sketch'
string3 = '''When Cleese says:
  "Now that's what I call a dead parrot" '''
```

1. In `string1`, the string is started with a double quote. As a result, Python interprets the apostrophe in *Python's* as part of the string and the quotation marks after *Circus* as the end of the string.
2. In `string2`, the string is started and ended with a single quotation mark. As a result, Python considers the double quotes around *"Dead Parrot"* as part of the string.
3. Strings enclosed in single and double quotes are *generally* limited to one line (remember the EOL error above?). Strings enclosed with triple quotes (single or double) can span multiple lines, for example:

```
# Try it!
string3 = '''When Cleese says:
  "Now that's what I call a dead parrot" '''
print(string3)
```

1.1 Integers and floating point numbers: Basic types for numerical data

There are two main numerical types:

- `int` for integers (zero, positive whole numbers, and negative whole numbers)
- `float` for all non-integer numbers.
- Integers are created by simply typing in a number without a decimal point (an integer literal).
- Floating point numbers are created by typing a number that contains a decimal point. Hence, in Python, `1` is `int` type, but `1.0` and `1.` are both of type `float`.
- Negative numbers are preceded by a negative sign.
- Scientific notation is also permissible (e.g., `6.02214076e23`) and *always* results in a `float`.
    - e.g., one million could be created as `1e6`. While the significand `1` is specified without a decimal point, the overall expression is still treated as a floating point number resulting in the Python representation `1000000.0`.

1.1 Integers and floating point numbers: Basic types for numerical data

The two numeric types serve different purposes:

- Integers can be represented perfectly by computers.
- However, computers have difficultly representing numbers of an arbitrary precision.
    - e.g, an irrational number like Pi (π) cannot be represented by a computer because it does not terminate.
    - Hence, floating point numbers must truncate the digits at a particular point as a way to *approximate* true values.

In general, if you are working with whole numbers, you should make sure to use integers. But be careful:

- Python may convert an `int` to a `float` if it believes such a conversion is necessary.
- Operations that convert between types representing the same underlying value are often called "casting".
- Addition, subtraction and multiplication of two integers always yields another integer.
- However, **division of integers will be cast to float** (`4/2 --> 2.0`).

1.1 Comparison of numerical values

It is okay to compare integers because they are not approximated internally. However, the approximation used to represent floating point numbers makes comparison of floating point numbers tricky. For example,

```
i = 1
test = i == 1   # --> True
print(test)
```

However,

```
f = 1 / 2.00000000000000000001
test = f == 1 / 2       # --> True
print(test)
```

While close to one-half, 1/2.00000000000000000001 is not equal to 1/2. This is an artefact of the approximation when working with floating points. ***Always* be careful when checking floating point numbers for equality.**

1.1 Booleans: A type for True/False data

Booleans (or `bool`) are another basic type, which are used for logical testing. Booleans are generated by a class called `bool`. This class can only generate two instances, called `True` and `False`. You can simply construct a `bool` in Python using literals.
Remember that literals are succinct ways of generating instances of a certain type. You can generate the two boolean instances using the characters *True* and *False*, without quotes and in title case.

> *Note: Title case refers to words with a leading capital letter and all remainder letters in lower case:*

```
True        # --> boolean representing true
False       # --> boolean representing false
```

True and *False* literals cannot be used as variable names:

```
True = 'a true value'
```

The exception raised is similar to the one we got when we tried `1 = x`:

```
    True = 'a true value'
    ^
SyntaxError: cannot assign to True
```

1.1 Booleans: A type for True/False data

However, you can create a variable called *true* (in lower case):

```
true = True
```

Boolean objects are often used to conditionally execute a statement. In other words, to execute a statement only if something is either `True` or `False`.

- For example, you may want to update your stock data weekly after the markets close on Friday. Thus, you can write code to check the day of the week and the time, executing code to download data only after the Friday market close.

1.1 Booleans: A type for True/False data

Expressions in Python will always generate an object as a result. Some expressions generate booleans as a result. This can be very useful when constructing tests.
Inequality tests:

```
# Less-than and less-than or eqaul tests
1 < 2     # --> `True`
2 < 2     # --> `False`
2 <= 2    # --> `True`
# Greater-than or greater-than or equal tests
1 > 2     # --> `False`
2 > 2     # --> `False`
2 >= 2    # --> `True`
```

But what if we want to test equality instead?

```
1 = 1
```

A single equal sign is used in assignment statements. To test equality, use `==` instead:

```
# Equality tests
1 == 2 # --> `False`
2 == 2 # --> `True`
```

1.1 Booleans: A type for True/False data

The `bool` type also permits several logical operations:

- `not` to flip `True` to `False` and vice versa.
- `and` to check that two Boolean values are both `True`.
- `or` to check if at least one of two Boolean values is `True`.

Here are some examples:

```
# not
not True        # --> False
not False       # --> True

# and
True and True   # --> True
True and False  # --> False
False and True  # --> False
False and False # --> False

# or
True or True    # --> True
True or False   # --> True
False or True   # --> True
False or False  # --> False
```

1.1 How can I check the type of an object?

To examine the *type* of any object: `type(object)` built-in function.

- This function can be applied to an instance or to a variable pointing to an instance.

```
y = type(1)   # --> <class 'int'>
print(y)
```

1. The expression `type(1)` will check the type of the integer literal "1". The output of this function is then assigned to `y`.
2. The output of the `print(y)` states the type of `y` is `<class 'int'>`.

What does `<class 'int'>` mean?

- Remember that classes are factories that produce instances. The class that produced the instance `1` is called `int`. This is the class that produces all integer instances.

We can change our example slightly as follows:

```
x = 1
y = type(x)
print(y)
```

1. the `int` instance is created, stored in the memory, and then assigned to the variable `x`.
2. Python will first fetch the instance referenced by the variable `x` and then check its type.
3. The result, `<class 'int'>`, is stored in a variable called `y`.

1.1 How can I check the type of an object?

The example below illustrates the difference between the integer literal `1` and the float literal `1.0`:

```
i = 1
type(i) # -> <class 'int'>

f = 1.0
type(f) # -> <class 'float'>

type(1.) # -> <class 'float'>
```

Here are some other examples using primitive types:

```
x = 1
print(type(x))          # <class 'int'>

xstr = '1'
print(type(xstr))       # <class 'str'>

test = x == xstr        # --> False
print(test)
print(type(test))       # <class 'bool'>
```

1.1 The `NoneType`

Before we proceed, also be aware of another important type called `NoneType`. This type produces a single instance, called `None`. The `None` instance is used to represent a null value, which is the absence of a value.

- It is important to keep in mind that `None` does not mean `0`, `False`, or an empty string. `None` represents a null value, that's it.

In general, operations between `None` and other basic types is not permitted (because it does not make sense).
How can you add the value `2` to a null value (remember that `None` is not zero)?

```
2 + None    # --> TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

1.1 The relation between types and classes

You may wonder why we are making such a big deal about classes and instances. These are crucially important concepts in Python because they are fundamental to understand how your code operates.
Consider the following two statements:

```
x = 1    ###create 'int' instance
xstr = '1'   ###create 'str' instance
```

The internal representation of the integer 1 is very different from the internal representation of the string "1". Consequentially, `int` instances and `str` instances have fundamentally different functionality.
Two integers can be added, but you cannot add a string to an integer or float:

```
# Try it!
'3' + 2 # --> TypeError: can only concatenate str (not "int") to str
```

This will result in yet another type of exception, called a `TypeError`. These exceptions are raised when an operation or function is applied to an object of inappropriate type. Also, Python tells you that you can only concatenate a `str` to another string.

1.1 The relation between types and classes

If you try to execute `'3' + '2'`, Python will assume you want to concatenate the strings "3" and "2", which will produce the string "32".

```
# You can concatenate strings with the "+" operator
print('3' + '2') #  output is "32"
```

What if you revert the order of the "sum" and try to execute the statement `2 + '3'`?

```
# Try it!
2 + '3' # --> TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Again, Python will raise a `TypeError` exception. However, the error message you receive is different: `TypeError: unsupported operand type(s) for +: 'int' and 'str'`. It is important that you understand why this is a different error message than the one above.
Remember that Python evaluates statements from left to right.

1.1 The relation between types and classes

What about the `*` operator? We can multiply numeric types using `*`, but what happens when we try the following:

```
# Try it!
'3' * 2   # --> '33'
```

This will *not* produce an error! The reason is the following: Python will interpret the statement `'3' * 2` as an instruction to "copy" the string `'3'` twice, which will result in the string "33". The fact that the string `'3'` represents a number has no importance:

```
# Try it
print('x' * 2) # output 'xx'
```

A big part of learning Python involves understanding how it will interpret the statements you write.

- This *syntax* can be tricky to master at first because some of the rules Python uses may not seem obvious (or logical).
- For now, just be aware that operators like "+" and "*" will be interpreted differently depending on the type of objects they operate on.

1.2 Interacting with objects using methods and attributes

Let's go back to our initial example:

```
# 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)
```

We now understand that statements (2), (3), and (4) are assignment statements which assign instances to variables `tic`, `start`, and `end`. What about statement (5)?

1.2 Interacting with objects using methods and attributes

```
df = yfinance.download(tic, start, end)   # (5)
```

- (5) is also an assignment statement which assign *some* object to a variable called `df`. This object is the result of the expression `yfinance.download(tic, start, end)`. To understand what it does, we first need to discuss *attributes* and *methods*:
- An attribute is a value associated with an object which is referenced by name using the "dot" notation. Roughly speaking, an attribute is a reference to an object that "belongs" to another object.
- A *method* is a function which is defined inside a class. Simple, yes?
    - Not really... There is a lot to unpack here. For starters, what do we mean by an object "belonging" to another? Enter namespaces...

1.2 Namespaces

A *namespace* is a place where variables are defined.

- e.g., the functions `print` and `type` are defined in the `builtins` namespace and always available.

One way to think about namespaces is the following:

- When Python "sees" a variable, it will search for the value associated with the this variable. The places Python will search for these variables are called namespaces.

Consider this statement:

```
x = 1
print(x)
```

1.we define the variable `x`. Every time we define a variable, a map will be created in *some* namespace. This map will have the form `name --> object` and will tell Python that the `name` refers to the `object`. So in the first statement, a map will be created in a namespace, which we will call *global* for now.
2.Python needs to find the object that is assigned to the variable `x`. It will search the *global* namespace and see that this namespace contains the map `x --> 1`. It will then replace the reference `x` with the instance `1`, which is what is printed.

1.2 Namespaces

The following will result in an exception:

```
print(x)   # --> NameError: name 'x' is not defined
x = 1
```

The reason for the exception is that Python did not find a map between the variable `x` and an object *in any namespace* it searched.
But how many namespaces are there and how does Python prioritise them?

- For now, lets focus on the two namespaces we mentioned: The *global* and *builtins* namespaces.
- Python will search for the variable `x` in the *global* namespace first. The *builtins* namespace is always searched last.

What about the following statement?

```
# Try it
print
```

Nothing will be printed if you execute the code above but no exception will be raised either, i.e., Python does not complain that the variable `print` is not defined.

- Although it is true that we did not define `print` in the *global* namespace, it *is* defined in the *builtins* namespace. Once Python finishes searching the *global* namespace, it will move to the *builtins* namespace.

1.2 Namespaces

There is a number of functions defined in the *builtins* namespace. You can find the full list of Python built-in functions **here**.
Many functions which are available in Python session are not defined in the *builtins* namespace. To understand how that works, let's take another look at the `str` class.
Consider the following statement:

```
x = 'abc'
```

We know by now that the string literal `'abc'` is a succinct way of instructing Python to generate a `str` instance. In a way, this is just a shortcut. What Python is actually doing is instructing the `str` class to generate an instance with the value "abc". Instances of classes are created by "calling" the class (just like a function). This means that the statement above is equivalent to:

```
x = str('abc')
```

To summarise: strings are generated by the class `str`. This class is defined in the `builtins` namespace so it is always available. To generate a `str` instance with value "abc" we call the class `str` using "abc" as a parameter, or `str('abc')`.

1.2 Namespaces

Now, imagine you would like to convert this string from lower to upper case. Wouldn't it be great if Python provided a function called `upper`? Unfortunately, `upper` is not a built-in function:

```
x = str('abc')
upper(x)    # --> NameError: name 'upper' is not defined
```

However, this function does exist in Python. The reason the code above failed is because we did not tell Python where this function is defined, i.e., we did not use the correct namespace.
Classes (the factories of instances) generate their own namespace. This namespace contains a map between all functions and variables defined *inside* the `str` class.
To understand how that works, lets go back to our Excel analogy.

- Pretend there is a class called `Excel` defined in the `builtins` namespace. What this class does is to produce instances of the `Excel` type. Instances of the `Excel` will automatically open the actual Excel application. So calling `Excel("some_file.xlsx")` will simply open the file *some_file.xlsx* in Excel.
- Suppose the `Excel` class defines another function, called `import_csv`. What this function does is to import the contents of a CSV file and then create the `Excel` instance. Because this `import_csv` function is defined by the `Excel` class, it belongs to the `Excel` namespace. To access the `import_csv` function, use its *qualified name* - `Excel.import_csv`.

1.2 Namespaces

A *qualified name* is a "dotted" name showing the “path” to the namespace. In the example above, `Excel.import_csv` tells Python that the function `import_csv` is defined in the `Excel` namespace.
Similarly, the `str` class defines a function called `upper`. To access this function, we need to refer to its qualified name: `str.upper`

```
x = str('abc')
xup = str.upper(x)
print(xup)          # --> 'ABC'
```

To summarise:

- Namespaces tell Python where to look for variables and functions.
- Variables and functions defined in the *global* and *builtins* namespace do not have to be name-qualified.
- `str` is a class defined in the *builtins* namespace, so we can call it directly. However, if we want to access functions defined inside the `str` class, we need to use qualified names.

1.2 Using methods to work with strings

Functions that are defined inside classes are called *methods*. So `upper` is a method of the class `str`. We can gain access to this method by calling it as an *attribute* of the class.
The distinction between *attributes* and *methods* can be quite confusing. To keep it simple, lets just agree that an attribute is how we access objects associated with other object. This object can be anything, including a method. So `upper` is a method that can be accessed as an attribute of `str`.
As we saw, we can create uppercase versions of the string "abc" by calling `str.upper('abc')`.
Here is where things get interesting (and complicated...):

- The `upper` method will be available to *any* instance generated by the class `str`.
- When we call a method as an attribute of an instance, the method will get the instance as its first attribute.

This means that we can create an uppercase version of "abc" in two ways:

```
x = str.upper('abc')
print(x)            # --> 'ABC'
y = 'abc'.upper()
print(y)            # --> 'ABC'
```

The important thing to understand is that the method `upper` is a method defined by the `str` class but which can be accessed as either a class attribute or an instance attribute:

```
str.upper   # --> class attribute
'abs'.upper # --> instance attribute
```

1.2 Using methods to work with strings

When we access the method as a class attribute, we need to pass a string as a parameter:

```
str.upper('abc')   # --> 'ABC'
```

However, when we access the method as an instance attribute, the instance will be passed as the first parameter to the method. Intuitively, think of `'abc'.upper` as a shortcut to `str.upper('abc')`.

```
'abc'.upper()   # --> str.upper('abc') --> 'ABC'
```

In the example above, the `upper` method followed a string literal. This does not have to be the case. We can create the instance first, assign it to a variable called `x` and then call `x.upper`.
Here is another example:

```
weird_case = "My fUnNy tYpEcAsE sTrInG"
weird_case_upper = weird_case.upper() # --> "MY FUNNY TYPECASE STRING"
```

1.2 Using methods to work with strings

The `upper` method is not the only one defined by the `str` class (see complete list of `str` method **here**). For example, the `str` class also defines a `lower` method, which you can use to convert strings to lowercase:

```
weird_case = "My fUnNy tYpEcAsE sTrInG"
weird_case_lower = weird_case.lower()     # --> "my funny typecase string"
```

Methods that generate strings can be chained together. For example, **instead of**:

```
weird_case = "My fUnNy tYpEcAsE sTrInG"

# First convert the string to uppercase:
weird_case_upper = weird_case.upper()       # --> "MY FUNNY TYPECASE STRING"

# Then, convert the uppercase version to lowercase
weird_case_lower = weird_case_upper.lower() # --> "my funny typecase string"
```

You could do

```
weird_case = "My fUnNy tYpEcAsE sTrInG"

# Convert the string to upper case and then to lower case
weird_case_lower = weird_case.upper().lower() # --> "my funny typecase string"
```

- `weird_case.upper().lower()` - Because `weird_case.upper()` returns a `str` instance, and the method `lower` is available to any `str` instance, we can call the `lower` method after calling the `upper` method.

1.2 Using methods to work with strings

Important to note that `weird_case.lower` and `wierd_case.lower()` return very different objects.

- The parentheses are critical. Typing `weird_case.lower` asks Python for the method referenced by the *attribute* `lower` belonging to the instance. It doesn't ask Python to execute/implement that method!

But why would you ever need access to the method?

- All we care about is the result from implementing the method, right? Not really... as we will see many times during this course, it is often useful to assign *methods* to variables, so they can be run later. For example:

```
weird_case = "My fUnNy tYpEcAsE sTrInG"
weird_case_fn = weird_case.lower    # --> <built-in method lower of str object at xxxx>
print(weird_case_fn())   # --> "my funny typecase string"
```

How is that possible?

- Do not forget: Everything in Python is an object. The method referenced by the `lower` attribute is also an object. These methods can be assigned to variables just like any other object. This ability to assign functions to variables can be used powerfully but is an advanced technique that we will return to later in the course.

1.2 Method calls with parameters

If you look at the list of *string* methods in the **Python standard library reference guide**, you will see that many methods have words between the parentheses.

- The *upper* and *lower* methods do not have them, indicating that these functions may be called with empty parenthesis `'abc'.upper()`.
- However, methods with words between parentheses indicate that the method accepts *parameters*.
- Take another look at the string methods in the **Python standard library reference guide**. It is important to understand how these methods are documented. You will notice the following patterns in the way parameters are documented:

```
str.some_method(req_parm)
str.some_method(req_parm[, opt_parm])
str.some_method(opt_parm=default_value)
str.some_method(*args, **kargs)
```

1.2 Method calls with parameters

As you can see, some consist of simply a name. Others have an equals sign following the name followed by a value. Still others are surrounded by brackets. Each of these annotations serves to tell us something about the parameter:

- **Required** parameters are simply listed as a name. These are **not** surrounded by square brackets, nor do they have an equals sign followed by a value. You must provide a value for this parameter when using the method.
- **Optional** parameters are enclosed in square brackets. These may be provided when calling the function but not required.
- **Default** parameters are listed with an equals sign followed by the default value. When you provide a value for this parameter it will be used and the default is ignored. If no value is provided for this parameter, the default value is used.
- **Multiple value** parameters are preceded by one or two stars. The difference between the one- and two-star variations is important when writing the function but is not too important when *using* the function.

1.2 Method calls with parameters

For example, the `str.center` method (note the US spelling) used to centre text is listed as `str.center(width[, fillchar])`. The terms `width` and `fillchar` indicate that this function takes up to two parameters. The first, `width`, indicates the total number of characters in the line inclusive of the `string` itself. So, to centre the word *Hi* in a line of 40 characters:

```
"Hi".center(40)  # --> '                    Hi                    '
```

- There are 19 characters before and after `Hi`. Thus, the total line width is 40 characters, as we requested.

Parameters enclosed in square brackets, like `fillchar` above are optional. If omitted, they take a default value. The default for *center* is to use a space. To center the word "Hi" inside dashes, simply add a dash string to the method call.

```
"Hi".center(40, '-') # --> '--------------------Hi---------------------'
```

Function parameters must be specified in order. The `center` method expects a line width followed by the `fillchar`. Calling it as `center('-', 40)` will create an error.

1.2 Formatting strings

Often, one wants to mix text and data together in a print statement to monitor the progress of a program.

- A simple way to do this is to use an `f-string`. Placing an `f` before a string instructs Python to replace any variable name found inside curly braces with the value of the variable.
- Thus, `f'{var}'` yields a string with the *value* of `var`, eliminating the curly brackets in the process. This can be used to provide easily understood data about a program.
    - e.g., if we have the variables a and b, we can write out their string representations as:

```
a = True
b = 5
print(f"The value of a is {a} and the value of b is {b}")
```

1.2 Formatting strings

Alternatively, you may use the format method of the string function to substitute values. In the simplest form, you place curly braces in the string {}, call the format and then supply the variables you wish to print in order.
The previous example could be written as:

```
a = True
b = 5
print("The value of a is {} and the value of b is {}".format(a , b))
```

The format method provides a large degree of customisation, the details of which can be found in the official Python documentation or on any number of online tutorials such as **pyformat.info**.

1.2 Summary

A method is simply a function defined inside a class.
We can call methods using attributes. When we call a method using the attribute of an instance, the instance will be passed as the first parameter to the method.
You can think of class attributes as giving you access to variables defined by the class namespace.
We mentioned that we needed to learn more about attributes and methods to explain statement (5) the 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)
```

It should be clear by now that `download` is a function defined by `yfinance`, which can be accessed using `yfinance.download`. But where does the `yfinance` come from?

- The (module) `yfinance` is not defined in the *builtins* namespace. As such, it can only be used after it is imported into the *global* namespace. This is what we are doing in statement (1). We will learn more about *modules* and *packages* later!

1.3 Variable Names: Rules and conventions

- Python gives the programmer a large degree of latitude when naming variables. However, there is a distinction between what you are *allowed* to do and what you *should* do.
- In this section we will take a closer look at naming rules (i.e., what you are allowed to do) and commonly accepted best practices for naming variables (i.e., what you should do).

1.3 Naming rules revisited

Previously, we discussed some important naming rules in Python:

1. Names must start with a letter or underscore ("_").
2. Names can only contain alphanumeric characters and underscores.
3. Names are case-sensitive.

All these rules apply to any variable, function, or class we create.
Importantly, there are some keywords that Python keeps to itself. The following identifiers are used as **reserved words** - cannot be used as ordinary identifiers:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

These *reserved keywords* will result in a syntax error, for example,

```
import = 'some string'
# This will raise the following exception:
#     import = 'some string'
#            ^
# SyntaxError: invalid syntax
```

1.3 Naming variables: Cautionary practices

Certain variable names should be avoided, even though they do not violate the Python syntax.
Built-in function names You can create variables with the same name as some of Python built-in functions.

- Some important function names that are not reserved keywords. If you assign an object to a built-in function name, you will override the built-in function with the object assigned to that variable.

1.3 Naming variables: Cautionary practices

For example, the following is permitted but highly discouraged:

```
# This is VERY bad...
str = 'this is a string'
```

This is because `str` is a Python built-in function used to generate strings. By creating a variable called `str`, you are overriding the built-in function `str` and the function is no longer available using the standard syntax.

```
# Create a variable containing the string "5"
x = str(5)
print(x) # --> '5'

# Create a variable called "str", overwriting the built-in method
str = "very bad idea!"

# Trying to use the function `str` again will raise an exception
x = str(5)

# Will raise the exception:
#     x = str(5)
# TypeError: 'str' object is not callable
```

1.3 Naming variables: Cautionary practices

To prevent a clash, the convention is to add an underscore to the name of the built-in.

```
# This is totally fine...
str_ = 'this is a string'

# No clash...
x = str(5)
```

1.3 Naming variables: Cautionary practices

Double leading underscores

Remember that you can start variable names with an underscore:

```
_var = 1
print(_var) # --> 1
```

- Python programmers typically reserve this style when creating variables or functions that should be viewed as *private*. The concept of private variables is beyond the scope of this course.
- That said, you will often find variables and functions starting with a single underscore in publicly available Python code. That typically means that the variable should not be accessed outside the scope in which the variable is defined.
- *Our recommendation*: avoid naming variables with a leading underscore in your code for now.
    - By *convention*, variables defined with a single underscore indicate that the variable is intended for "internal use".

1.3 Naming variables: Cautionary practices

Double leading underscores

Leading double underscores often have special meaning to the Python Interpreter.

- In fact, Python will typically treat variables starting with a double underscore very differently than other variables. This is *not* a convention, it is part of the syntax.
- One example of such a variable is `__name__`, which we will discuss later in the course. Unless you know exactly how Python will interpret names which start with double underscores, you should not *create* variables (or functions) with double leading underscores.

1.3 Naming variables: Best practices

Over time, the Python community has evolved these conventions into style guidelines. The style guide can make your code cleaner, more elegant and easier to read. The most important style guidelines are described in the **PEP 8 - Style Guide for Python Code**, or "PEP 8" for short.

- You can view PEP 8 as your starting point for developing your own style, and be consistent.

For example, when it comes to variables, the style guideline suggests:

- *Standard variables:* Use *snake case* which is written using *lower-case* letters with words separated by underscores, e.g., `my_standard_variable`.
- *Constant variables:* Use *screaming snake case* which is written using *upper* case letters with words separated by underscores, e.g. `MY_CONSTANT_VARIABLE`.
- *Class:* use title-case variable names which is written using *a capital letter followed by lower-case letters*, e.g., `Variable`.

1.4 Adding functionality: Modules and packages

As programs get bigger, one often needs to:

1. Split the code into multiple files
2. Incorporate functionality that has been written by other people.

- For example, a large program that contains 10,000 lines of code would be difficult to work with in a single file. Splitting into multiple files allows you to reflect the underlying logic of your code in the files. It also allows you to use certain portions of your code in other programs.
- But how to incorporate code written outside the current Python file?

1.4 Modules and packages

Each file of Python source code technically defines a *module*. PyCharm's default Python project includes a file called `main.py`. Thus, the code in this file resides in a module called *main*.
If you were working on a Finance project, you might group code related to stock tickers in a file called `ticker.py`. The code in this file would reside in a module called *ticker*. Each module's code is distinct. Code in `ticker.py` would not be automatically accessible from `main.py`.
More generally, a module is an object that serves as an organisational unit of Python code. Any file containing Python code is thus a module because it groups pieces of code together. Modules have a namespace containing all objects defined in the module.

- To get access to these objects, we need to **import** the module into Python.

Modules can contain any object, including other modules. Python modules that contain other modules are called *packages*. Like any module, modules defined inside a package have their own namespace, which is nested into the namespace of the package.

- To get access to a module defined inside a package, we also need to **import** the package first.

1.4 Modules and packages

Consider the `yfinance` package, which is not part of the Python standard library. We installed this package using PyCharm's package manager.
Intuitively, what PyCharm did was to create a folder somewhere in your computer called `yfinance` and include all modules that belong to this package inside this folder **yfinance link**:

```
yfinance/
|__ __init__.py
|__ base.py
|__ multi.py
|__ shared.py
|__ ticker.py
|__ tickers.py
|__ utils.py
```

Each *.py* file is a *module*. The folder `yfinance` is a package because it contains modules and a special file/module called `__init__.py` . Among other things, this special module is telling Python to treat the folder `yfinance` as module.
How do we access objects defined in a given module?

1.4 The import statement

Returning to our Yahoo! Finance 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)
```

As we previously discussed, statement (5) indicates that the function `download` can be accessed as an attribute of the module `yfinance`. In other words, the function `download` belongs to the namespace `yfinance`. Modules and packages define namespaces, which is how Python will gain access to the objects they define.

All the statements above are defined in the *global* namespace. We can get access to objects defined in the *global* namespace and in the *builtins* namespace. We also mentioned that classes define their own namespaces.

- e.g., the reason we can use the method `str.upper` is because we are telling Python the function open is defined in the *builtins* namespace `str`.

1.4 The import statement

The `yfinance` module is not part of the *builtins* namespace. This means that this module is not immediately available and needs to be imported first.
The `import` statement is used to "load" modules.

```
- create a namespace which Python will use to access the other objects defined inside this module.
```

Intuitively, `import yfinance` instructs Python to do the following:

1. Find a module called `yfinance`. The folder `yfinance` contains a file called `__init__.py`. As such, Python will treat this folder as a package.
2. Since `yfinance` is a module, it will have a namespace. This namespace will contain the module `__init__`. This means that `yfinance.__init__` is a reference to the module `__init__` inside the `yfinance` module.
    - Note that the module name is `__init__` but the file is `__init__.py`.
3. Python will then evaluate all statements inside the file `__init__.py`. Any object defined in the `__init__.py` file will be added to the `yfinance` namespace. For instance, if the `__init__.py` file contained a statement like `a = 1`, then `yfinance.a` would return `1`.

`__init__` module will add the function `download` to the `yfinance` namespace. For now, just imagine that the function `download` is "defined" inside the `__init__.py` file (we will cover what actually happens a bit later).

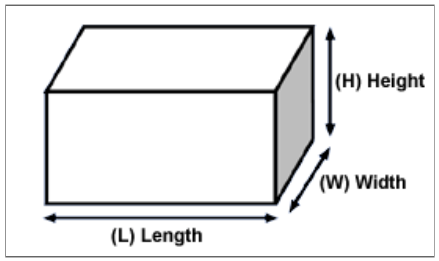
1.4 Summary

- Typically, *import statements* will be at the top of every Python file you write.
- These statements allow you to incorporate code written in *modules* outside the current file. Such modules may be code written elsewhere in your project.
- Typically, this module will be a file (e.g., `some_module.py`) or a folder called *some_module* containing a file called `__init__.py`.
    - If the module is a file, all the statements inside this file will be executed.
    - If it is a folder, all statements inside the `__init__.py` will be executed.

Later in the course, we will learn how to create our own modules and packages.

In-class Exercise 1

Please write a Python code to compute volume of a box, given the length is 65 cm, width is 36.2 cm, and height is 28.5 cm.



```python
### Sample Solution
length = 65
width = 36.2
height = 28.5
volume = length * width * height
print(f"the volume of the box is {volume} cubic centimeters.")
```

2.0 What are data structures?

- Nearly every program you write will require working with groups of items. In a stock downloader, you will probably want to have a `list` of stock tickers to download. When working with price data, you may want to deal with the bid, midpoint and ask prices. This means that a single price quote, may contain multiple information like bids and asks in the form of a `tuple`. Python `sets` allow us to track things, ensuring that we only have one of each object. This can be useful to ensure we don't do computations twice, for example.
- Data structure can also refer to maps between keys and values. If you are working with corporate financial data, you will probably want to know earnings before interest and taxes (EBIT) for a variety of firms. In such cases, you probably want to have a way to easily look up the EBIT for a specific firm. One way to do this is via a Python `dictionary`.

2.0 What are data structures?

The Python standard library has four commonly needed data structure classes, and each has particular uses in general programming:

- Lists: The `list` class implements a collection of objects which can be ordered. This collection is **mutable** - elements can be added or removed from it. Lists can contain duplicates, e.g., `[1, 1]` is a list containing two elements with identical value. Since lists are ordered, we can refer to its elements by their position in the list. The elements of a list can be any Python object.
- Tuples: The `tuple` class also implements a collection of objects. Similarly to lists, objects are ordered and can be accessed by their position in the collection. Duplicates are also allowed. Unlike lists, `tuple` are **immutable** -- Once a tuple is created, you cannot add elements to it or remove elements from it.
- Sets: A `set` is collection of objects without natural ordering. This collection is **mutable** but does not support duplicates, e.g., you cannot create the set `{1, 1}` in Python. If you try, Python will only consider one instance and return `{1}` instead. Unlike lists and tuples, not all Python objects can be elements of a set.
- Dictionaries: The `dict` class implements collections of that map each element of a set (called a *key*) to a certain Python object (called a *value*). Like sets, dictionaries cannot contain duplicate keys.

2.1 Lists - Intro

The `list` class is critically important in Python. Python lists contain multiple objects in a specific order. Lists are easy to construct. Lists start with an opening square bracket, `[`, include a sequence of elements separated by commas, and then end with a closing square bracket, `]`. For example,

```
lst = [1, 2, 3]
print(lst)

t = type(lst) # --> <class 'list'>
print(t)
```

You'll note that the `list` class does not indicate that the list has integers. You are free to mix types within the list:

```
lst = [1, "string", True, None]
print(type(lst)) # --> <class 'list'>
```

2.1 Lists - Intro

As mentioned previously, a `list` is not a primitive type because it is not a `string`, `int`, `float`, or `bool`. Lists may contain any objects, even other lists:

```
lst_a = [1, "string", True, None]
lst_b = ["a" , lst_a]
print(lst_b)
```

You can create an empty list by using square bracket `[]` or `list()`.

```
lst = []
lst = list()
```

2.1 Lists - Intro

Importantly, an empty list is *not* a list containing `None`.

```
empty_lst = []
not_empty_lst = [None]
```

When we studied the `str` class, we pointed out that string literals are convenient ways of "calling" the `str` method. Similarly, using square brackets are convenient ways of calling the `list` method, which is responsible for creating instances of the `list` type.

- For example, the following statements are equivalents:

```
lst = [1, 2, 3]
lst = list([1,2,3])
```

2.1 Lists - Indexing

How do we access elements inside a list?

- Typically, we use brackets to tell Python we want to get an element from a collection. For lists, we refer to elements by their position in the list. Like most programming languages, Python uses **zero-based indexing** where the initial element in a list is given item number `0`.
- After the initial item at index `0`, each subsequent element in a list is indexed by the next relevant integer. So, for example, in the list `lst = [1, "string", True, None]`:
    - The element `1` is at position 0. We can access this element using `lst[0]`.
    - The element at position 1 is `"string"`, which we can get using `lst[1]`.
    - Similarly, `lst[2]` refers to `True` and `lst[3]` to `None`.
- Indexing beyond the last element of a list will result in an `IndexError`, indicating that the you tried to access beyond the end of the list.

2.1 Lists - Indexing

In the statements below, we use *f-strings* to print out the element of the list.

- Remember that strings literals that start with *f* will replace the objects inside `{}` with the string representation of their values:

```
lst = [1, "string", True, None]
print(f"The item at index 0 is {lst[0]}")
print(f"The item at index 1 is {lst[1]}")
print(f"The item at index 2 is {lst[2]}")
print(f"The item at index 3 is {lst[3]}")
```

The following will result in an exception:

```
lst = [1, "string", True, None]
print(f"The item at index 4 is {lst[4]}") # --> IndexError: list index out of range
```

2.1 Lists - Indexing

Zero-based indexing starts with the initial element of the list. You may also index the list "backwards" from the last element to the first using negative numbers. In that case, the last element of the list is given index `-1`, with each element to the left further into the negative integers.
So, for example, in the list `lst = [1, "string", True, None]`:

- `lst[-1] --> None`
- `lst[-2] --> True`
- `lst[-3] --> "string"`
- `lst[-4] --> 1`

```
lst = [1, "string", True, None]
print(f"The item at index -4 is {lst[-4]}")
print(f"The item at index -3 is {lst[-3]}")
print(f"The item at index -2 is {lst[-2]}")
print(f"The item at index -1 is {lst[-1]}")
```

Negative indexing beyond the first element of a list will result in an `IndexError`, indicated that the you tried to access beyond the start of the list.

```
lst = [1, "string", True, None]
print(f"The item at index -5 is {lst[-5]}") # --> IndexError: list index out of range
```

2.1 Lists - Slicing

- While individual elements can be accessed by their position, *slices* can be used to access subsets of the list. *Slices* are created by using square brackets and listing the two indices separated by a colon.
- Python will create a slice from the object at the start index through the object **before** the ending index.
- For example, take the list `lst = ["a", "b", "c", "d", "e", "f"]`. `lst[1:4]` returns *a list* consisting of elements starting at index `1` through the element at index `3` (not 4). In other words, the slice is `["b", "c", "d"]`.

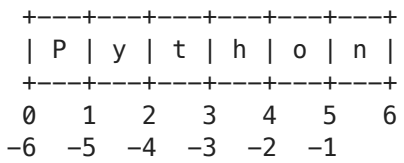
```
lst = ["a", "b", "c", "d", "e", "f"]
print(f"The slice from index 1 through 4 is {lst[1:4]}" )
```

- You may also use negative indexing as you see fit. Thus, in the example above, `lst[1:4]` is equivalent to `lst[-5:-2]`.

```
lst = ["a", "b", "c", "d", "e", "f"]
print(f"The slice from index -5 through -2 is {lst[-5:-2]}" )
```

2.1 Lists - Slicing

- This decision to take a slice up to, but not including the final index is a bit counter-intuitive. However, the way to remember that Python follows this rule is to recognise that the length of the slice may be calculated as the difference between the end and start index. So, the slice `lst[1:4]` creates a slice of `4 - 1 = 3` elements (the elements at index 1, at index 2, and at index 3).
- One way to remember how slices work is to think of the indices as pointing between characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of n characters has index n.
    - The following example (taken from the Python documentation) illustrates:



2.1 Lists - Slicing

- You may also indicate that you want to take a slice from the beginning by omitting the first index.
    - For example, `lst[:4]`. This is equivalent to `lst[0:4]`.
- You may also indicate you want to take a slice to the end of a list by omitting the final index.
    - For example, `lst[1:]` takes a slice from the item and index 1 through the end of the list.
- What happens when we try to create slices beyond the indexes in a list?
    - Perhaps surprisingly, this will *not* result in an exception. One way to think about this is the following: Every time you ask for a slice, Python will return the *intersection* between the indexes:

```
lst = ["a", "b", "c", "d", "e", "f"]
print(f"lst[:3] gives {lst[:3]}")
print(f"lst[0:1000] gives {lst[0:1000]}")
```

2.1 Lists - Commonly used methods

Lists are created by the `list` class. This class also defines a number of functions, which are called `list` methods. Similar to `str` methods, you can access these functions using class attributes or instance attributes.
Consider the `append` method. This method append an element to the end of a list. The length of the list increases by one.

```
lst_a = [1]             # lst_a is [1]
list.append(lst_a, 2)   # lst_a is now [1, 2]
print(lst_a)
```

Like string methods, list methods can be accessed using instance attributes. In that case, the instance will be passed as the first parameter to the function:

```
lst_a = [1]       # lst_a is [1]
lst_a.append(2)   # lst_a is now [1, 2]
print(lst_a)
```

2.1 Lists - Commonly used methods

Although in the statements above `list.append(lst_a, 2)` and `lst_a.append(2)` will have the same effect, `lst_a.append(2)` is much more common. In general, you should use instance attributes to access list methods.
It is important to note that the `append` method will modify the list in place. It does *not* return a new list!
Another important list method is called `extend`. This method will extend the list (in place) with the elements of a second list.

```
lst_a = [1]
lst_b = [2, 3]
lst_a.extend(lst_b) # --> lst_a now is [1, 2, 3]
print(lst_a)
```

2.1 Lists - Commonly used methods

Remember that lists can be nested. This means that you can append a list to another list.

```
lst_a = [1]
lst_b = [2, 3]
lst_a.append(lst_b) # --> lst_a now is [1, [2, 3] ]
print(lst_a)
```

You can also insert elements to a list at specific locations, using the method `insert`.
You may also place an element at a specific location using the `insert(i, x)` method. In this case, `i` indicates the element location you would like the new object `x` to be inserted. In other words, object `x` is insert before the object currently at position `i` in the list:

```
lst = [1, True, None]
print(f"lst is originally {lst}" )
lst.insert(1, "string")       # lst is now [1, "string", True, None]
print(f"After insertion, lst is now {lst}")
```

2.1 Lists - Commonly used methods

There are three methods to remove objects from a list:

- `remove(x)` eliminates the first object in the list with value `x`. It will result in an error if `x` is not in the list.
- `pop()` (without parameters) will remove the last object in a list; `pop(i)` will remove the object at index `i`.
- `clear()` will remove all elements in the list.

```
lst = [1, "string", True, None, True]
print(f"Original lst is {lst}")

lst.remove(True) ### take a look of the output, 1 will be removed.
print(f"lst after removing the first True is {lst}")

lst.pop(2)
print(f"lst after removing the element CURRENTLY at index 2 is {lst}")

lst.pop()
print(f"lst after removing the CURRENT last element is {lst}")
```

2.1 Lists - Commonly used methods

Lists may be reversed using the `reverse` method. You may also sort elements using the `sort` method. Sorting follows a *natural* order of the elements. Numbers will be sorted in ascending order, and strings will be sorted alphabetically, with all capital letters considered to be before lower-case letters -- in other words, `Z` is before `a` .
Finally, the built-in `len` function will return the length of the list.

```
lst = [1, "string", True, None, True]
no_of_items =len(lst)
print(f"lst has {no_of_items} items")
```

Python has a rich set of `list` methods **official documentation**.

2.1 Lists - Split function

The split() method splits a string into a list. You can specify the separator, default separator is any whitespace.
Syntax: string.split(separator, maxsplit)

- separator (Optional): Specifies the separator to use when splitting the string. By default any whitespace is a separator.
- maxsplit (Optional): Specifies how many splits to do. Default value is -1, which is "all occurrences".

```
line = "welcome to the class"

x = line.split()

print(x)
```

2.2 Tuples - Intro

Tuples are a very important container in Python. A tuple is simply a sequence of objects. It is declared within parentheses, and the object elements are separated within the tuple using commas. To declare a tuple that contains the objects `"word"`, `5` and `False`, in that order, use the following statement:

```
t = ("word", 5, False)
```

Just like with `lists`, you may create empty tuples two ways:

1. provide a starting parenthesis followed immediately by a closing one, `()`;
2. with `tuple()`. However, empty tuples have limited use in real-world Python.

2.2 Tuples - Indexing and Slicing

Accessing individual elements of a tuple works just like lists. Use square bracket zero-based indexing (the first element has index 0). In the example above, `t[0]` is `"word"`, `t[1]` is `5`, and `t[2]` is `False`. Negative indexing is supported with `t[-3]` being `"word"`, `t[-2]` being `5`, and `t[-1]` being `False`. You may also get a slice of a tuple using the colon notation used with lists.

```
t = ("word", 5, False)

print(f "The item at index 0 is {t[0]}")
print(f "The item at index 1 is {t[1]}")
print(f "The item at index 2 is {t[2]}")

print(f "The item at index -3 is {t[-3]}")
print(f "The item at index -2 is {t[-2]}")
print(f "The item at index -1 is {t[-1]}")

print(f "The slice from index 0 through 2 is {t[0:2]}" )
```

2.2 Tuples - Packing and Unpacking

Unpacking is a way we can elegantly get the objects out of the tuple container without needing to index each individual item. To unpack a tuple, list the variable names on the left side of the assignment operator and the tuple on the right. For instance:

```
# Create a tuple with two elements
tup = (1, 2)

# unpack the tuple into two variables:
(a, b) = tup

# print
print(f"`a` is {a} and `b` is {b}")
```

Python actually does not care whether you use parentheses when packing (creating) or unpacking tuples. If omitted, Python will act as if all parentheses were present. We used parentheses above to make it clear we were unpacking Tuples.
The following lines would be treated equivalently by Python:

```
a, b, c = 1, True, "word"
a, b, c = (1, True, "word")
(a, b, c) = 1, True, "word"
(a, b, c) = (1, True, "word")
```

2.2 Tuples - Packing and Unpacking

Tuples have an additional advantage in Python in tracking down bugs. The length of the Tuple begin unpacked on the left side of the assignment operation must be equal to the length of the Tuple that is packed on the right hand side. If you try to use the wrong number of variables, Python will throw an error.

```
(a, b) = (1, True, "word")        # --> ValueError: too many values to unpack (expected 2)
(a, b, c, d) = (1, True, "word")  # --> ValueError: not enough values to unpack (expecte d 4, got 2)
```

With one exception, Python will not throw an error if you incorrectly unpack to a single variable. In such cases, the single variable is assigned the value of the entire tuple.

```
a = ( 1 , True , "word" ) # a is (1, True, "word")

# Explicit parentheses does not force Python to count variables being unpacked
(a) = (1, True, "word") # a is (1, True, "word")
```

2.2 Differences between tuples and lists

- Lists and tuples are both with sequences of elements. However, tuples and lists differ in one key aspect.
    - Lists are fundamentally mutable, meaning they can change. As mentioned previously, it is possible to add and remove elements from a list using the append and pop functions, respectively.
    - In contrast, tuples do not let you add or remove elements. The objects a tuple refers to are fixed when the tuple is created. There is no way to add or remove elements---there is no tuple method equivalent of list's `append` and `pop` functions.

2.3 Sets - Intro

Sets are collections that ensure each object appears only once.

- For example, if you want to download stock data for 20 stocks, you can simply create a *List* of the tickers. In this case, if a ticker appears in this list twice (because of some silly mistake), your code will download the data for that ticker twice. Conversely, even if you try to include the same ticker multiple times in a set, the underlying object will contain a single copy of each element.

Sets are constructed using curly braces: type an opening curly brace `{`, a comma-separated list of items, and a closed curly brace `}`. Python will ensure that the set only contains one of each element.

```
s = {1, 2, 3, 3, 3, 3} # --> {1, 2, 3}
```

Unlike *Lists* and *Tuples*, you **cannot** create an empty set using curly braces `{}`. This is because this literal constructs an empty dictionary (which we will cover next).
To create an empty set, you must use the `set` method, `set()`.

```
empty_set = set()
```

2.3 Sets - Commonly used methods

The most commonly used methods for sets involve adding and removing elements: `add` and `remove`. In the example below, we create an empty set and then add tickers to it.

```
s = set()
s.add("QAN.AX")   # s is {"QAN.AX"}
s.add("WES.AX")   # s is {"QAN.AX", "WES.AX"}
s.add("CBA.AX")   # s is {"QAN.AX", "WES.AX", "CBA.AX"}
s.add("CBA.AX")   # s is {"QAN.AX", "WES.AX", "CBA.AX"} (so no change)

print(f"After adding objects, s is {s}")

s.remove("CBA.AX") # s is {"QAN.AX", "WES.AX"}
print(f"After removing 'CBA.AX', s is {s}")
```

The example above illustrates an important feature of *Sets*. We attempted to add the `str` object with the value `"CBA.AX"` twice. However, the *Set* ensures we only have one such object in it. When we request to remove that object, the *Set* removes that single instance.

2.3 Sets - Commonly used methods

It is often helpful to check if an object is in a *Set* or not. To do so, Python has the `in` operator. The expression `object in a_set` will return `True` if `object` is in the set `a_set`, and `False` otherwise.
Conversely, you can check if an object is *not* in a set by using the *not in* operation. That is, `object not in a_set` will return `True` if the object is *not* in the set, and `False` if the object is in the set.

```
s = {1, 2, 3}
1 in s # --> True
4 in s # --> False

1 not in s # --> False
4 not in s # --> True
```

You can check the number of objects in a *Set* using the `len()` built-in function.

```
s = {1, 2, 3}
n_of_elements = len(s)
print(f"s contains {n_of_elements} objects")
```

2.3 Sets - Differences from *Lists* and *Tuples*

Sets do not support indexing or slicing.
Mathematically, *Sets* are not intrinsically ordered.

- For example, you may think of a *Set* of numbers having an order in terms of value (e.g., 1 occurs before 2, which occurs before 3). Alternatively, you may consider the ordering of a set to depend on the order each object is inserted.
- But, it is not obvious if you add an element twice to a set if the set should order the element at its first addition or its second. For example, if we add `1,` , `2,` and `1` to a set, in that order, we have two possible ways to consider the set order. The set could be considered `1` then `2` (ordering based on the first insertion of an object) or `2` then `1` (ordering based on the last insertion of an object).
- This is in contrast to *Lists* and *Tuples*, which are naturally sorted. The items in both are placed in order and may be accessed using zero-based indexing. When you add an object to a *List*, you need to specify its precise location.

Python does not allow you to index nor slice into a *Set*.

- As we'll see later, the current version of Python keeps track of insertion into a *Set* using the first insertion approach from above, which is used when looping over the objects in a *Set*. However, previous versions (e.g., Python 2) does not track insertion order; the objects in a Python 2 *Set* may appear in any order when looping, regardless of when they were inserted.

In general, it's best practice to consider your sets unordered.

2.4 Dictionaries - Intro

- *Dictionaries* are the most complicated of the Python's basic data structure. Before going in to its technical features, let's consider where the term gets its name by comparing them to actual dictionaries.
- A dictionary is, obviously, a reference work that contains definitions of words. Each word is in the dictionary exactly once, and linked directly to a definition. However, definitions may appear more than once (e.g., synonyms).
- Python dictionaries have similar features:
- While real dictionaries link words and definitions, Python dictionaries link *keys* and *values*.
- The *word* from an actual dictionary is represented by an *key* object that can appear exactly once.
- The *value* object need not be unique and multiple *keys* can have the same *value*.
- Dictionaries serve as a sort-of hybrid between sets and lists. Like sets, each *key* appears once. But, like lists, we can have a variety of object values.

2.4 Dictionaries - Creating *Dictionaries*

A list of daily stock prices is a natural fit for a dictionary (or, as we'll see later, a Pandas *DataFrame*). Consider the following table, representing stock prices for a number of days in 2020:

Date	Price
'2020-01-02'	7.1600
'2020-01-03'	7.1900
'2020-01-06'	7.0000
'2020-01-07'	7.1000
'2020-01-08'	6.8600
'2020-01-09'	6.9500
'2020-01-10'	7.0000
'2020-01-13'	7.0200
'2020-01-14'	7.1100
'2020-01-15'	7.0400

2.4 Dictionaries - Creating *Dictionaries*

Let's create a dictionary that associates each date in "Date" with the corresponding price in "Price". In other words, we want a "map" from the elements in the columns "Date" to those in "Price". That will allow us to fetch prices by their dates (but not the other way around). The table below summarises the relationship:

Keys	Values
'2020-01-02'	7.1600
'2020-01-03'	7.1900
'2020-01-06'	7.0000
'2020-01-07'	7.1000
'2020-01-08'	6.8600
'2020-01-09'	6.9500
'2020-01-10'	7.0000
'2020-01-13'	7.0200
'2020-01-14'	7.1100
'2020-01-15'	7.0400

2.4 Dictionaries - Creating *Dictionaries*

We'll create a dictionary corresponding to this stock price table above:

```
prc_dic = {
  '2020-01-02': 7.1600,
  '2020-01-03': 7.1900,
  '2020-01-06': 7.0000,
  '2020-01-07': 7.1000,
  '2020-01-08': 6.8600,
  '2020-01-09': 6.9500,
  '2020-01-10': 7.0000,
  '2020-01-13': 7.0200,
  '2020-01-14': 7.1100,
  '2020-01-15': 7.0400,
  }
print(prc_dic)
```

2.4 Dictionaries - Creating *Dictionaries*

Syntactically, we create the dictionary using curly braces `{}`. We then list key-value pairs. Each key is separated from its value by a colon, and each key-value pair is separated by commas. While this is the same as with *Sets*, separating the key and the value using a colon, `:` indicates to Python that we want a dictionary. The order matters! The key will be on the left of ':', and the value on the right.
Note that our dictionary definition spans multiple lines. This is permissible when creating containers and makes it easier to read.
Once we create a dictionary, we can look up or "query" each value by its key (or label). We use `[key]` to look up values:

```
prc_dic = {
  '2020-01-02': 7.1600,
  '2020-01-03': 7.1900,
  '2020-01-06': 7.0000,
  '2020-01-07': 7.1000,
  '2020-01-08': 6.8600,
  '2020-01-09': 6.9500,
  '2020-01-10': 7.0000,
  '2020-01-13': 7.0200,
  '2020-01-14': 7.1100,
  '2020-01-15': 7.0400,
  }

# Ask for the value stored under '2020-01-02'
prc_dic['2020-01-02'] # --> 7.16
```

2.4 Dictionaries - Creating *Dictionaries*

Dictionaries are one way maps (`key --> value`). The following will result in an exception:

```
prc_dic = {
  '2020-01-02': 7.1600,
  '2020-01-03': 7.1900,
  '2020-01-06': 7.0000,
  '2020-01-07': 7.1000,
  '2020-01-08': 6.8600,
  '2020-01-09': 6.9500,
  '2020-01-10': 7.0000,
  '2020-01-13': 7.0200,
  '2020-01-14': 7.1100,
  '2020-01-15': 7.0400,
  }

prc_dic[7.1600] # --> KeyError: 7.16
```

The `KeyError` indicates that we tried to access the value under a key that does not exist.

```
- The fact that we cannot get the *key* associated from the *value* makes sense when you think in terms of real-
world dictionaries. These do not have an easy way to get the word associated with a definition. And, as many
words may share definitions, it's not even clear which word to use.
```

2.4 Dictionaries - Creating *Dictionaries*

You can create an empty *dictionary* in two ways:

1. provide an initial open curly brace followed by a closing one: `{}`.
2. use the `dict()` function.

```
d = {}      # --> {}
d = dict()  # --> {}
```

Like the other containers, you can count the number of *key-value* pairs in a *dictionary* using the `len` function. Since each *key-value* pair has a unique *key*, this container size is equivalent to counting the number of unique keys in the *dictionary*.

- *Note*: This does not count the number of unique *values*, nor does it count the total number of *keys* and *values* (i.e., number of *keys* plus the number of *values*).

```
dic = {'a':1 , 'b':1}
number_of_keys = len(dic)
print(f"dic has {number_of_keys} keys")
```

2.4 Dictionaries - Adding, changing, and removing key-value pairs

One way to adding a key-value pair to a dictionary is to use the assignment: `dictionary[key] = value`. On the left-hand side, we have our way to index into the dictionary. By assigning, Python will associate the value with the key. Python is intelligent enough to do this even if the key is not already in the dictionary:

```
# Initialise the dictionary
prc_dic = {
  '2020-01-02': 7.1600,
  '2020-01-03': 7.1900,
  '2020-01-06': 7.0000,
  '2020-01-07': 7.1000,
  '2020-01-08': 6.8600,
  '2020-01-09': 6.9500,
  '2020-01-10': 7.0000,
  '2020-01-13': 7.0200,
  '2020-01-14': 7.1100,
  '2020-01-15': 7.0400,
  }

# Store the next date in the dic
prc_dic['2020-01-16'] = 6.9200

print(f"prc_dic now has a value for 2020-01-16: {prc_dic}")
```

2.4 Dictionaries - Adding, changing, and removing key-value pairs

You can change the *value* associated with a *key* by assignment as well. Since a dictionary only has one *value* for each *key*, this will overwrite the *value* currently associated with the *key* :

```
d = {"a": 1}
d["a"] = 2
print(f"After overwriting, d is now {d}" )
```

The dictionary `update` method also allows for modifying a dictionary in place. The update method takes either a mapping object (e.g. another dictionary) or a sequence (e.g. a list or tuple) of (key, value) pairs. `update` replaces the values of existing keys and creates entries for new keys:

```
d_update_with_dict = {'a': 1, 'b': 2}
d_update_with_dict.update({'a': 2, 'c': 3 })
print(f"After update d_update_with_dict is {d_update_with_dict}")

d_update_with_list = {'a': 1, 'b': 2 }
d_update_with_list.update([('a', 3), ('c', 5 )])
print(f"After update d_update_with_list is {d_update_with_list}")
```

2.4 Dictionaries - Adding, changing, and removing key-value pairs

Removing a pair from a dictionary is done using the `pop` dictionary method. As with looking up a *value* in a dictionary, removing a pair is done by specifying the *key*, but the method removes both the *key* and the *value*.

```
d = {'a': 1, 'b': 2 }
d.pop('a')
print(f"After popping 'a', d is now {d}")
```

2.4 Dictionaries - Notes on dictionary keys

Duplicate keys on creation

You may wonder what happens if you accidentally duplicate a key when creating a dictionary. Python will not complain if you repeat a *key* with different *values*:

```
dic = {'a' : 1, 'a': 2}
print(dic)
```

As indicated previously, Python will simply override duplicate keys and store a single value for each key. You should not try to guess which value will be selected because nothing guarantees the algorithm will remain the same in future releases (in Python 2, the order of the pairs did not matter, in Python 3 it does).

Similarly, you can duplicate the entire *key-value* pair. This should also be avoided

```
dic = {'a': 2, 'a': 2, 'a': 2, 'a': 1 , 'a' : 1}
print(dic)
```

2.4 Dictionaries - Notes on dictionary keys

Hashable keys

Dictionary keys must be *hashable* objects. For our purposes, that means keys must be immutable objects, that is, objects whose instances cannot be modified. This means we can use:

- All primitive types, which are immutable. For example, we cannot modify a string instance in place. String methods like `replace` will always generate new instances.
- Tuples, which are immutable in that the objects referred to within the tuple can not be changed. However, mutable objects in a tuple may be changed. For example, you can add elements to a *list* that is contained with a *tuple*.

However, the *Lists*, *Set*, and *Dicts* containers are all mutable. You can add, remove, and change objects within them. Thus, these cannot be used as dictionary keys.

The good news is that if you make a mistake and try to set a dictionary key to a mutable object, Python will not allow.

2.4 Dictionaries - Are dictionary ordered?

Up until recently, elements in a dictionary were not ordered at all. Starting with version 3.6, the order of insertion will be preserved. However, **you should avoid thinking of dictionaries as ordered** (even though they are now).
Insertion order is only one way of thinking about how the dictionary was created. We think of dictionaries in terms of the keys. If our keys are dates, then the dates have a natural chronological order. This is not necessarily the order the dictionary was created, and, as a result, relying on how Python orders the dictionary is not a good idea. It will become a major source of confusion if you do.
Even though dictionary preserve the order elements are inserted, you cannot refer to elements by their position:

```
# After Python 3.6, order of insertion is preserved
dic = {1: 'first', 2: 'second', 3: 'third'}
# This will insert the 0 key at the end
dic[0] = 'fourth'
# This will NOT insert the key at the end because 1 exists
dic[1] = 'new first'
# Also, this does not return the first element of the dic
# because it was inserted at the end
dic[0]
```

In-class Exercise 2

Please write a short program to get the email domain name of the following example.

- 'From firstname.surname@unsw.edu.au Tue Oct 06 10:10:15 2020'

```
### Sample Solution
line = 'From nickname.surname@unsw.edu.au Tue Oct 06 10:10:15 2020'

domain = line.split()[1].split('@')[1]

print(domain)
```