Toolkit for Finance

Week 3: Control Flows

Lecturer: Dr. Yiping LIN

Overview

- *Additional Topic: Assignment statements and copies*
- Control Flows
    - Sequential (*default*)
    - **Conditional Executions**
    - **Loops**
- More Control Flow (*Next Week*):
    - *Functions*
    - *Comprehensions*

Recap: Data Structure Quick Comparision

1. List - mutable, ordered
    - `[ ]` or list( )
2. Tuple - unmutable, ordered
    - `( )` or tuple( )
3. Set - mutable, unordered, no duplicate
    - Only set( )
4. Dictionary - mutable, *unordered*, key-value pair
    - `{ }` or dict( )

Recap: List Slicing

Basic Structure:

- lst[start:stop] # items start through stop-1
- lst[start:] # items start through the rest of the array
- lst[:stop] # items from the beginning through stop-1
- lst[:] # a copy of the whole array

lst[1:] vs. lst[1:-1]?

- lst[1:] will give you the slice from the second item till the end;
- lst[1:-1] will return the slice from the second item till the second last item.

A. Assignment statements and copies

Previously, we learnt how assignment statements can be used to assign or re-assign instances to variables (e.g., `a = 1`) and to modify elements of mutable objects (e.g., `dic[key] = value`). In some cases, you can even use an assignment statement to modify the value of an attribute.
In general, these statements take the following form:

```
<target> = <expression>
```

In the expression above, `<target>` is a *name* that refers to some object in the computer memory.

- For example, the statement `var = 'a'` assigns a `str` instance with value "a" to the (variable) name `var`. After this statement is evaluated, every time the interpreter sees the variable `var`, it will fetch the `str` instance "a" in the computer memory.

A. Assignment statements and copies

Now, consider the following statements:

```
var1 = 'a'
var2 = 'a'
var3 = 'a'
var4 = 'a'
```

How many `str` instances with value "a" are stored in the computer memory?

- The answer is one! In other words, there is a single `str` instance with value "a" stored in the computer memory and all variables `var1`, `var2`, `var3`, and `var4` refer to the *same* instance in the computer memory.

To understand why, ask yourself the following question: Would it make any sense to create multiple copies of the `str` instance "a"?

- No, because the `str` instance "a" cannot be changed. Like all primitive types, `str` instances are not mutable. If they cannot be changed, there is no reason to keep multiple copies of it in the computer memory.

A. Assignment statements and copies

Lets try to understand this concept by looking at mutable objects instead.

```
var1 = ['a']
var2 = ['a']
var3 = ['a']
var4 = ['a']
```

Now, there will be four distinct `list` instances in the computer memory. Each one of these instances will (initially) contain the same `str` instance "a".

- Python keeps all four instances in the computer memory is because lists are mutable. We can change some of these instances without affecting the others. For example, `var1.append('b')` will modify the instance associated with the variable `var1`, but will not affect the other lists.

It is natural to think that each one of these list instances contains its own copy of the `str` instance "a".

- This is *not* true! In fact, `list` instances in the computer memory do not "contain" other objects. What they contain are ***references*** to other objects. This means, intuitively, the list `['a']` is actually stored as `[address of instance 'a']`, which refers to the "address" of the instance "a" in the computer memory.
- **When we modify a list, we are actually modifying the objects referenced by the list, not the list itself.**

A. Assignment statements and copies

To see that, consider the following example:

```
lst1 = ['a']
lst2 = ['b', lst1]
lst3 = ['b', ['a']]
```

It may look like `lst2` and `lst3` are different ways of creating the same list, namely, the list `['b', ['a']]`.

- This is *not* the case. `lst2` is *not* creating a copy of `lst1`. Instead, when the interpreter reaches `lst1` in the expression `['b', lst1]`, it will simply substitute `lst1` with the address of the list `['a']` created in the first statement. This means that if we modify the second element of `lst2`, we will also modify `lst1`!

```
lst1 = ['a']
print(f'This is lst1: {lst1}')

lst2 = ['b', lst1]
print(f'This is lst2: {lst2}')

lst2[1].append('c')
print(f"This is lst2 after modifying it: {lst2}")
###This is lst2 after modifying it: ['b', ['a', 'c']]

print(f"This is lst1 after modifying lst2: {lst1}")
###This is lst1 after modifying lst2: ['a', 'c']
```

A. Assignment statements and copies

Conversely, modifying `lst3` will *not* change `lst1`:

```
lst1 = ['a']
print(f'This is lst1: {lst1}')

lst3 = ['b', ['a']]
print(f'This is lst3: {lst3}')

lst3[1].append('c')
print(f"This is lst3 after appending 'c': {lst3}")
###This is lst3 after appending 'c': ['b', ['a', 'c']]

print(f"This is lst1 after modifying lst3: {lst1}")
###This is lst1 after modifying lst3: ['a']
```

Those examples above illustrate two important features of compound objects:

1. Python will create multiple copies of mutable objects in the computer memory, even if they have the same value.
2. If a mutable object contains a reference to another mutable object already stored in the computer memory, Python will *not* automatically copy the latter.

1.0 Control Flow: Intro

- Control flow is the computer science term to describe how programs execute steps. As an imperative language, Python requires that programmers explicitly describe how program execution proceeds.
- Python's default process is to execute one statement at a time, proceeding from the top of a program file to the bottom (*Sequential*). As we'll see, control flow statements allow us to alter this natural flow, repeating certain statements, skipping others, and moving around within and across files.
    - Control flow, therefore, allows the program to implement logic and adapt to input or to the current state of computations.
- We start this discussion with conditional execution, a technique that allows us to execute code depending on whether certain logical conditions are satisfied.
    - This serves as the basis for analysing loops and functions later.

1.1 Conditional Execution - if statements

The most basic of conditional execution statements is the `if` statement. It indicates a block of code to be executed if a condition holds.

- An `if` statement consists of the `if` keyword, followed by a logical expression, and a colon.
- After that statement, we indent all code to be run if the logical expression holds.

The basic structure of an `if` statement in pseudo-code is:

```
if logical_expression:
    statement_a
    statement_b
    ...

statements outside the if statement
```

1.1 Conditional Execution - if statements

Let's look at the following example:

```python
happy = True
if happy is True:
    print("I am happy")
print("This will print regardless of the value of happy")
```

1. Python assigns the `bool` instance `True` to the variable `happy`.
2. `if happy:`, Python first checks if the expression `happy is True` holds. Since, `happy` is a variable, Python will substitute this variable with the instance assigned to it -- the `happy is True` statement becomes `True is True` instance.
    - `is` operator will test if two instances *are the same*. The `bool` class defines only two instances, `True` and `False`. As such, `True is True` will evaluate to be `True`, meaning that the condition is satisfied.
3. If the condition is satisfied, Python will then print the string "I am happy".
4. Python considers the `if` statement closed when it finds a line that is not relatively indented. Thus, it resumes normal execution with the second print statement.

1.1 Conditional Execution - if statements

```python
happy = True
if happy is True:
    print("I am happy")
print("This will print regardless of the value of happy")
```

Try changing `happy` to `False` in the example and see what happens if you rerun the code.
Two observations above:

1. the `print("I am happy")` statement is indented. **Indentation is of critical importance in Python.**
2. the values of `happy` in the example are Boolean values: `True` or `False.` You may wonder what would happen if we set happy to something else (feel free to try values such as `"False"`, `0`, or `[]`).

1.1 Conditional Execution - White space

Code under the `if` statement is indented.

- Blank areas in code are called white space due to how they would appear printed on paper. While white space is optional in some languages, it is required in Python.

White space is used in Python to delineate that statements form an interrelated block of code.

- In conditionally executed code, all the code that should be executed when the logical condition holds must be indented under the `if` statement. Python looks at this code and will execute each statement only if the condition holds.
- Once Python finds a statement that is not indented relative to the `if` statement, it resumes normal execution regardless of whether the logical condition in the if statement held.

1.1 Conditional Execution - White space

The standard in Python is to indent each code block with four spaces. Indented code blocks can be nested - use four spaces to indent a code block inside another indented code block.

```python
happy = True
very_happy = True

if happy is True:
    print("I am happy")                         # <-- four-spaces indentation
    if very_happy is True:                      # <-- four-spaces indentation
        print("Actually, I'm really happy!")    # <-- eight-spaces indentation
```

1.1 Conditional Execution - White space

We used the term *space* and not *tab*, spaces are the standard.

- However, typing four spaces to indent code is undesirable. Therefore, most Python text editors are configured to insert four spaces each time you press the tab key.
- `PyCharm` and `Ed` both do this automatically for you: Pressing *tab* inserts four spaces. Pressing delete on the character immediately following the 4^th^ space will remove four spaces. But the underlying characters in the code are spaces, not tabs.

Python doesn't enforce the white space standard described above. You are free to use any number of spaces, as long as you are consistent. The only requirement is that all interrelated code is indented at the same exact level.

- If the first statement after an `if` statement is indented two spaces, then all subsequent statements must be indented two spaces.
- The fact different standards are allowed does not mean that you should consider them. Deviating from the "four-spaces" standard will likely lead to serious and unintended consequences.

1.1 Conditional Execution - Truthy/Falsy is just bad-y

Conditionally executed code should only be run *if* a condition holds. These either/or conditions are best modelled using Boolean logic values of `True` and `False`. However, Python does not require that you use Boolean values inside `if` statements:
For example, consider the following code:

```python
happy = True
if happy:
    print("I am happy")
```

It may seem like `if happy` is equivalent to `if happy is True`.

- This is *NOT* the case! Try the following:

```python
happy = 5
if happy:
    print("I am happy")
```

1.1 Conditional Execution - Truthy/Falsy is just bad-y

This will still print the "I am happy" statement, even though we set `happy = 5`. Obviously, Python is not checking if `happy` refers to the `bool` instance `True`. If we go back to the previous version of the `if` statement example, nothing will be printed:

```python
happy = 5
if happy is True:
    print("I am happy")
```

The problem with `if happy:` it is not clear to Python what you are actually testing.

- This condition will only fail under certain conditions, e.g., when the value of `happy` is `None`, `False`, and empty string or list, 0, etc...

```python
happy = 0
if happy:
    print("I am happy") # <-- will not be printed
```

1.1 Conditional Execution - Truthy/Falsy is just bad-y

The value of 5 is not `True`, but Python treats it as if it is---it is True-ish (or Truthy), so How Python decides to treat non-Boolean values in `if` statements is sometimes called *Truthy/Falsy*?

- Python defines None, False, 0 numerical values, empty strings and empty containers from the standard library (list, dict, tuple, set) as False. All other standard types are treated as `True`. For classes outside the standard library, you would need to look at the documentation to decide what is `True` and `False`.
- We believe it is best practice to ensure that your logical tests work with the strict Boolean values of `True` or `False`. That is, `if` statements should use Boolean and not rely on Python's *Truthy/Falsy* rules.
    - Truthy/Falsy can create some issues with debugging when values are not of the expected class. In such cases, *Truthy/Falsy* coercion can allow code to execute that should not, and make it difficult to track down bugs.
- Importantly, even if you are simply checking if a variable is True, we recommend using the `is True` test. If you are checking a value is False, we recommend using an `is False` test.

1.1 Conditional Execution - Truthy/Falsy is just bad-y

The following table lists logical tests that can be used to explicitly test objects:

Object Type	Truth condition	Test
Object	Object is not None	x is not None
Boolean	True	x is True
Numeric	Non zero	x != 0
List	Non-empty	len(x) > 0
Dict	Non-empty	len(x) > 0
Tuple	Non-empty	len(x) > 0
Set	Non-empty	len(x) > 0

Please note that the length tests here for list, dict, tuple, and set, it can be slow for very large containers. We recommend refactoring your code to eliminate checks on non-empty containers.

1.1 Conditional Execution - Boolean logic

Recall:

The Boolean values of True and False are designed for constructing logical tests. There are three main operators for Boolean logic: `not`, `and`, and `or`.

- `not` is a simple Boolean operator. It turns a True value to a False and a False value to a True.

```
if not False:
   print("not False is True")

if not True is False:
   print("not True is False")
```

- `and` uses two Boolean values. If both are True, then you will get a `True` value, otherwise you will get a `False`.

Statement	Result
True and True	True
True and False	False
False and True	False
False and False	False

1.1 Conditional Execution - Boolean logic

- `or` also uses two Boolean values. If either one is True, then you will get a `True` value, otherwise you will get a `False`.

Statement	Result
True or True	True
True or False	True
False or True	True
False or False	False

`and` and `or` can chain to more than two Boolean values. If all the statements are of the same type (e.g. all `and`), it is straightforward to work through the logic by evaluating the first pair, using the result on the second and so on.
When mixing `and`, `or`, and `not` statements, it is best to use parentheses to indicate how the expression should be evaluated!!!
[Boolean-order]: Python evaluates `not` statements first, then `and` statements, and finally `or` statements.

1.1 Conditional Execution - Comparison operators

You will frequently need to compare two objects to determine if a condition is met. Fortunately, Python supports most of the comparisons you are used to from algebraic inequalities. You can compare two objects `a` and `b` using a number of operators.

- To test if `a` is equal to `b` use double equal-signs, `a == b` (single equals is reserved for assignment).
- You can also check if `a` and `b` are not equal using the not equals operator `!=` as `a != b`. Note that this is simply for convenience; you can always test for inequality using `not a == b`.
- Python also provides operators for strict and non-string inequality tests. To test that `a` is less than `b`, use `a < b`. Less than or equal to is done as `a <= b`. Greater than and greater than or equal to are `a > b` and `a >= b`, respectively.

These operators have intuitive meaning when applied to the numeric type `int`. **However, you should generally not test `floats` using any equality or inequality operators.**

- Floating point numbers are not represented exactly within Python. Instead, Python uses an approximation. Therefore, `a == b` applied to two floating points does not test if they are equal; it tests whether the approximations for `a` and `b` are equal.
- It is possible that the approximations are equal but the true values are not.

1.1 Conditional Execution - Comparison operators

These operators work with any object that supports them. To test for equality, the object's class must implement special methods (e.g. `__eq__` for equality and one of several possibilities for inequalities).
In such cases, you may compare object instances `a` and `b` and the interpretation depends on the class.

- For example, `strings` support alphabetical comparison. Words that come first alphabetically are "less" than those that come later. `"That" < "This"` will yield `True` because "That" comes before "This" alphabetically. Clearly, this implies that `"That" > "This"` is `False` by Python operators evaluation of string instances.

```
less_than_test = "That" < "This"
print(f'"That" < "This" yields {less_than_test}')
greater_than_test = "That" > "This"
print(f'"That" > "This" yields {greater_than_test}')
```

1.1 Conditional Execution - The difference between ”==” and ”is”

The operators `==` and `is` are *not* equivalent.

- `==` is called the *equality operator*. The `==` operator will compare the *value* of two objects;
- `is` is called the *identity operator*, which will compare if the objects correspond to the same instance in the computer memory.

Consider the following statements:

```
var1 = 'a'
var2 = 'a'
var3 = ['a']
var4 = ['a']
```

What's the relationship of var1&var2, var3&var4?

1.1 Conditional Execution - The difference between ”==” and ”is”

We mentioned previously, only one instance of the `str` "a" will be stored in the computer memory, whereas `var3` and `var4` are assigned to different instances. This means that:

```
var1 = 'a'
var2 = 'a'
print(var1 is var2) # --> True
print(var1 == var2) # --> True
```

Whereas:

```
var3 = ['a']
var4 = ['a']
print(var3 is var4) # --> False
print(var3 == var4) # --> True
```

Choosing between `==` and `is` can be quite difficult at first, so lets keep it simple:

- Use `is` **only** to test if something is either `None`, `True`, or `False`. For everything else, use `==`.

1.1 Conditional Execution - The `else` statement

So far, we've only used the `if` statement to run code when a certain condition is met.
In many cases, you want different statements to execute when the condition is not met. This is done using the `else` keyword and a second block. Indentation again is important here. The `else` statement should be indented as far as the `if` statement to indicate that they form a group.

```python
b = False
if b is True:
   print("b is True")
else:
   print("b is not True")
```

1.1 Conditional Execution - The `elif` statement

You may also have more than two blocks of code that you want to run under different conditions.
Multiple sets of conditions are specified using `elif` and a logical test.
Indentation is important again, with `if`, `elif`, and `else` statements at the same indentation level treated as a group.

```python
a = 0
b = True
if a != 0:
   print("a is non-zero")
elif b is True:
   print("b is True")
elif a < 0 and b is True:
   print("a is negative and b is True")
else:
   print("None of the conditions above hold")
```

1.1 Conditional Execution - The `elif` statement

Note that **Python will execute the first block that satisfies a condition**. That is, **any condition met later will not be executed** if a prior condition was satisfied.

- Repeating the code above with `a = -1` will trigger the first `if` condition only, even though the other two `elif` statement conditions are satisfied:

```
a = -1
b = True
if a != 0:
   print("a is non-zero")
elif b is True:
   print("b is True")
elif a < 0 and b is True:
   print("a is negative and b is True")
else:
   print("None of the conditions above hold")
```

1.1 Conditional Execution - The `pass` statement

Roughly speaking, the `pass` statement means "do nothing here". This means that nothing will happen when the interpreter evaluates the if statement below:

```
happy = True
if happy is True:
    pass
```

Why would we ever use the pass statement?

- One good reason is as a **placeholder** when developing code. e.g., when writing complicated logical tests, you can initially set up the conditions and place pass statements where you wish to later write code:

```
if x > 0 and y is True:
    # Write code for this case later
    pass
elif x <= 0 and y is True:
    # Write code for this case later
    pass
else:
    # Write code for this case later
    pass
```

An interactive program

- You can instruct Python to pause and read data from the user using the input( ) function:

  `input(prompt)`

- The input( ) function returns a string

```
name = input('Who are you?')
print('Welcome to the class,', name)
```

In-class Exercise 1

- Write a pay calculator code to compute weekly pay.
- When your weekly working hours is below or equal to 35 hours, the rate is 51.45 per hour, any hours above 35, the rate is 88.9.
- Using the input( ) function to let the user enter hours worked, and you can enter 50 hours as an example.

Hint: Remember input function returns string, to convert string to intger, you can use `int()`

```
In [ ]:
```

```python
### Sample Solution

hours = input('Enter number of hours you worked this week: ')
#Python 3 output numeric value as a string in input () function
hours = int(hours)
normal_rate = 51.45
overload_rate = 88.9

if hours > 35 :
    pay = (35 * normal_rate) + ((hours - 35) * overload_rate)
else :
    pay = hours * normal_rate

print(f'This weekly payment is: {pay}')
```

1.2 Loops - Intro

Previously, we saw how conditional statements can be used to perform actions depending on whether a condition is true or false.
Another useful control flow concept is **looping**.

- Loops allow us to execute code multiple times, changing key variables on each execution.

To see the value in looping, let's return to our stock price downloader.

- Suppose we would like to extend the Yahoo Finance download code to download multiple stocks. For example, we are also interested in Wesfarmers' stock prices ("WES.AX").
- The first option is to simply copy the code we already have, rearrange it a bit, and add statements to download Westfarmers' prices.

1.2 Loops - Intro

The program below will download, print, and then save both Qantas' and Wesfarmer's stock prices:

```
# Import the yfinance module
import yfinance
# Set the dates parameters
start = '2020-01-01'
end = None

# Download Qantas stock prices
tic = "QAN.AX"                          # (Q1)
df = yfinance.download(tic, start, end) # (Q2)
print(df)                               # (Q3)
df.to_csv('qan_stk_prc.csv')            # (Q4)

# Download Wesfarmers stock prices
tic = "WES.AX"                          # (W1)
df = yfinance.download(tic, start, end) # (W2)
print(df)                               # (W3)
df.to_csv('wes_stk_prc.csv')            # (W4)
```

Note that we separated the variables holding the start and ending dates because they are the same for both stocks.

- **But is this the recommended approach?**

1.2 Loops - Copy and paste: An anti-pattern

The answer is definitely **NO**...
In computer science, the term **"pattern"** typically refers to generally accepted solutions to common problems.
Conversely, the term **"anti-pattern"** refers to habits or responses to recurring problems that are considered ineffective, likely to lead to coding mistakes, or simply counterproductive. Anti-patterns should be avoided because these are often poor solutions to problems, which make your code harder to understand and maintain.
Copying and pasting code is generally considered to be an anti-pattern.

- Programming languages exist to make your life easier. They are capable of implementing complex logic.
- Copying code and then making minor changes means that you are manually embedding logic into the code instead of having the language help you.
- We should take a minute to think before copying and pasting code to determine whether we are implementing a good solution to a problem. Most often we are not.

1.2 Loops - Copy and paste: An anti-pattern

```
# Import the yfinance module
import yfinance
# Set the dates parameters
start = '2020-01-01'
end = None
# Download Qantas stock prices
tic = "QAN.AX"                              # (Q1)
df = yfinance.download(tic, start, end)     # (Q2)
print(df)                                   # (Q3)
df.to_csv('qan_stk_prc.csv')                # (Q4)
# Download Wesfarmers stock prices
tic = "WES.AX"                              # (W1)
df = yfinance.download(tic, start, end)     # (W2)
print(df)                                   # (W3)
df.to_csv('wes_stk_prc.csv')                # (W4)
```

Statements `(Q1)` to `(Q4)` are copied and pasted, then modified into statements `(W1)` to `(W4)`.
This may seem fine for two stocks, but it would be terrible for fifty.
Worse, what if we want to get more data than stock prices, like balance sheet data? What if we simply wanted to download the data (without printing it)? We will need to make updates for each individual ticker.

- These issues can be avoided by *refactoring* the code using `loops`. Refactoring is a process whereby we alter our code to make it more modular and easier to maintain and update.

1.2 Loops - For loops

Python has two primary looping constructs: `for` and `while.`
In a `for` loop, specific statements are repeated. In our case, we would like to "loop" over our tickers and execute the code to download the return data.
A `for` loop in Python takes the following form:

```
for <variable> in <iterable>:
    <statements>
```

where:

- The `for` keyword declares that you are implementing a *for loop*.
- `<iterable>` refers to an *iterable* object, that is, an object that will return its elements one at a time.
- `<variable>` is the name of the variable each element in `<iterable>` will be assigned to as we progress through the loop.
- `<statements>` refer to the indented code block containing all the statements that will be evaluated during each iteration of the loop.
- `in` is also a keyword and necessary in the loop statement. It indicates that each element "in" the `<iterable>` should be assigned to the `<variable>`, one at a time.

1.2 Loops - For loops over list

Many objects in Python are iterable, including all containers.
Lists are iterable. So we can create a list of tickers including "QAN.AX" and "WES.AX", and then loop over them:

```python
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')
```

In the `for` statement, Python identifies that we want to loop over the values in the list `tickers`.

- It does this sequentially. The first time through the loop, Python assigns the value of "QAN.AX" to the `tic` variable.
- It then downloads and prints the data. As in our initial script, we would like to save these data in a CSV file, and we do that in two steps:

1.2 Loops - For loops over list

1. we create a variable called `tic_basename`:

```
tic_base = tic.lower().split('.')[0]
```

- The expression `tic.lower().split('.')[0]` combines string methods and list indexes. If `tic = "QAN.AX"`, this statement is equivalent to:

```
tic = "QAN.AX"
# Convert the ticker to lower case
tic_base = tic.lower()              # --> 'qan.ax'

# Split the string into a list,
# using '.' as a separator
tic_base = tic_base.split('.')    # --> ['qan', 'ax']

# Fetch the first element of the list
tic_base = tic_base[0]            # --> 'qan'
```

1. We use an `f-string` to create the name of the CSV file. If `tic_base = 'qan'`, the name of the file will be `qan_stk_prc.csv`. For "WES.AX", the name of the file will be `wes_stk_prc.csv`.

1.2 Loops - For loops over list

- You do not have to use a list to store the tickers.
    - For example, sets and tuples would work as well because (like all basic containers) they are iterable.
- Iteration is a term used to designate things that can be repeated, like for loops. Thus, iterables are objects whose elements are returned one at a time.
- As with `if` statements, `for` loops define a code block. Python considers all lines that are indented relative to the `for` statement as part of the loop.
    - It executes these statements, changing the *variable_name* as appropriate for each pass through the loop.
- After concluding the `for`, Python finds the first statement that is not indented relative to the `for` statement and resume normal execution.
    - That is, Python considers lines up to, but not including, the first line that is not indented under the `for` statement as part of the loop.

1.2 Loops - Looping over Tuple

Below are examples of looping over all the containers we have seen so far:
Recall that tuples are a collection of values in Python quite similar to lists. Looping over tuples is just the same as looping over lists. Python begins with the first element (at index `0`) and then increments to each subsequent element in the tuple.

```
for v in ("string", True, 1):
    print(v)
```

1.2 Loops - Looping over Set

Sets are useful collections that allow you to ensure that each object appears only once. Given our desire to work with tickers, a Set is a natural data structure to use. This ensures that we analyse each stock once even if we make a mistake and add a stock to the set twice:

```
tickers = set()
tickers.add("QAN.AX")
tickers.add("QAN.AX")
tickers.add("WES.AX")

for tic in tickers:
    print(tic)
```

When looping over sets, you should not plan on receiving the elements in any specific order. Sets are fundamentally unordered containers.
While Python may deliver the elements in the same order every time in a loop, changing the elements or adding new class instances to a set can create some unexpected results.

1.2 Loops - Looping over Dictionary

Dictionaries support three different methods of looping depending on whether you want to loop over

1. keys,
2. values or
3. both keys and values simultaneously.

First, looping over keys is the default. On each pass, Python picks a different key to assign to the loop variable.

- As keys are unique, you are guaranteed to see each key object once. For example:

```python
d = {
  "beauty": True,
  "truth": True,
  "red wheelbarrow": 100000,
  5: "fingers",
  }
for key in d:
  print(key)
```

1.2 Loops - Looping over Dictionary

Second, You may loop over the values by calling the `values` dictionary method.

- On each pass through the loop, Python picks a value from the dictionary to assign to the loop variable. While keys are unique in dictionaries, values are not. Hence, you may see a value object multiple times.

```python
d = {
    "beauty": True,
    "truth": True,
    "red wheelbarrow": 100000,
    5: "fingers",
    }
for val in d.values():
    print(val)
```

1.2 Loops - Looping over Dictionary

Third, you may loop over a tuple packing the key and value as (key, value) using the method `items`.
The `items()` method returns a view object. The view object contains the key-value pairs of the dictionary, as *tuples in a list*.
Thus, each pass through the loop results in a tuple containing a different key and its related value packed in a tuple:

```python
d = {
    "beauty": True,
    "truth": True,
    "red wheelbarrow": 100000,
    5: "fingers",
    }
for key_value_tuple in d.items():
    print(f"key_value_tuple is {key_value_tuple}")
    # Unpacking
    key, value = key_value_tuple
    print(f"KEY: {key} VALUE: {value}")
```

1.2 Loops - Looping over different data types

Alternatively, you can simply unpack this tuple at the beginning of the loop:

```
d = {
    "beauty": True,
    "truth": True,
    "red wheelbarrow": 100000,
    5: "fingers",
    }
for key, value in d.items():
    print(f'KEY: {key} VALUE: {value}')
```

As with sets, you should not plan on receiving the keys or values in any specific order when looping over dictionaries. Conceptually, dictionaries are unordered containers. Python now returns dictionary elements in the order they were inserted but, again, it is not good practice to think of dictionaries as ordered collections.

1.2 Loops - Range

Some programming situations call for the use of sequences of integers.

- The built-in `range` function can be used to "generate" such sequences, which are called **range**.

There are two ways to call the range function, which can be quite confusing:

- The first version is `range(start, stop[, step])`, where:
    - `start` is the starting integer of the sequence.
    - `stop` is the upper bound for the sequence, meaning that the sequence will end with the largest integer `i` such that **`i < stop`**.
    - `step` is the integer that will be added to each preceding element in the sequence. The parameter `step` is **optional** and defaults to `1`.

1.2 Loops - Range

For example, to iterate over the integers 0 to 4:

```python
# omitting the `step` parameter
for i in range(0, 5):
    print(f"i is now {i}")
```

This is equivalent to:

```python
# Explicitly setting the `step` parameter to 1
for i in range(0, 5, 1):
    print(f"i is now {i}")
```

On each pass through the loop, Python simply picks the next value from the range up to the integer *immediately* before the stop value. Here, the first pass has `i` equal to 0, the second pass has `i` equal to 1, and so on.

1.2 Loops - Range

- The second version is `range(stop)`. This is essentially a shortcut to `range(0, stop)`.
    - `range(stop)` will return a sequence of integers from 0 to `stop-1`.
    - **The length of these sequence is `stop`** (because it starts at 0). This means that the following loop will go through five iterations:

```
for i in range(5):
    print("i is now {}".format(i))
```

You may be wondering why we use the method `format` in the statements above?

- The reason is that you will encounter such methods in many pieces of code, so we might as well provide you with some examples.
- To be clear, the `print` statement above is equivalent to `print(f"i is now {i}")`.

1.2 Loops - Range

You can use the `step` parameter to "skip" elements in a sequence of integers.

- For example, we can count over even numbers by setting the step value to 2:

```
for even in range(0, 10, 2):
    print("even is now {}".format(even))
```

When the `stop` value is not multiple of the steps away from the `start` value, Python simply stops at the last number it sees before the `stop` value.

- For example, we could ask Python to loop over a range from 0 to 9 in steps of 2. In this case, Python will stop at 8, which is the last even number before 9. Hence, the `range(0, 10, 2)` is equivalent to the `range(0, 9, 2)`:

```
for even in range(0, 9, 2):
    print("even is now {}".format(even))
```

1.2 Loops - Range

The step does not need to be positive. We can have a countdown by specifying a step of -1 and a start value *greater* than the ending value:

```python
for i in range(5,0):
    print("This will not execute because start is greater than stop.")

for i in range(5,0,-1):
    print("This message will self-destruct in {} secs...".format(i))
```

Make sure you understand:

1. why the first range without a step failed?
2. why the second countdown stopped at 1 instead of 0?

1.2 Loops - Enumerations

As you progress through an iterable, you sometimes want to keep track of the iteration number.

- In other words, often both the values and their positions in the iterable used inside the loop.

Consider the following example:

```
letters = ["a", "b", "c", "d", "e"]
print(f"letters = {letters}")
i = 0
for x in letters:
    print(f"letters[{i}] --> {x}")
    i += 1
```

You may wonder what is the mysterious statement `i += 1`?

- You can think of `var += expression` as a shortcut to `var = var + expression`. This means that `i += 1` is equivalent to `i = i + 1`.
- What we are doing here is creating a variable `i` outside of the loop and assigning it a value of 0. During each iteration, we are adding 1 to the value assigned to `i`.

1.2 Loops - Enumerations

Alternatively, you can use the built-in function `enumerate` to make this code more pythonic, `enumerate(iterable, start=0)`

- This function takes an iterable (from start which defaults to 0) as a parameter and returns another iterable.
- The returned iterable consists of tuples `(i, ele)`, where `i` is the current iteration number and `ele` is element in the iterable.

```python
letters = ["a", "b", "c", "d", "e"]
for tup in enumerate(letters):
    print(tup)
```

Instead of using the Tuple in the loop, most Python developers will unpack the two elements of the tuple immediately on creating the loop:

```python
letters = ["a", "b", "c", "d", "e"]
print(f"letters = {letters}")

for i, x in enumerate(letters):
    print(f"letters[{i}] --> {x}")
```

1.2 Loops - Enumerations

You are not restricted to apply the `enumerate()` function to a list; it works with any iterable object.

- For example, you can loop over the items in a dictionary along with their enumerated value as:

```python
d = {
    "beauty": True,
    "truth": True,
    "red wheelbarrow": 100000,
    5: "fingers",
    }
for i, tup in enumerate(d.items()):
    print(f'Iteration {i} gives {tup}')
```

1.2 Loops - Enumerations

You can pass a `start` parameter to `enumerate`. This allows you to choose the starting value of the `i` in the returned tuples `(i, ele)`.

- For example, the following will start the iteration counter at 1000:

```
d = {
    "beauty": True,
    "truth": True,
    "red wheelbarrow": 100000,
    5: "fingers",
    }
for i, tup in enumerate(d.items(), start=1000):
    print(f'Iteration {i} gives {tup}')
```

In-class Exercise 2

Random Number Test

- How does your brain work?

In-class Exercise 2

WRITE A CODE USING CONTROL FLOWS TO FIND THE LARGEST VALUE IN THE NUMBERS BELOW.

```
numbers = [3,9,1,5,7,2,11,0,3,12,3,15]
```

```python
### Sample Solution
numbers = [3,9,1,5,7,2,11,0,3,12,3,15]

temp_largest = None
print('Before', temp_largest)

for number in numbers:
    if temp_largest is None:
        temp_largest = number
    elif number > temp_largest:
        temp_largest = number
    print(number, temp_largest)
print(f'The largest value is {temp_largest}')
```

1.2 Loops - While loops

Python has another principal looping mechanism, the `while` loop. While loops run while a condition holds, and stop running as soon as the condition fails. The basic syntax in Python pseudo-code is:

```
while <condition_is_true>:
    <statements>
```

As with `for` loops, all statements indented one level from the `while` statement are treated as part of the `while` loop.

- For example, to sum up the integers less than or equal to 100, we could loop over a range. Since the range stops one before the last index, the range runs from 1 to 101:

```
the_sum = 0
for i in range(1,101):
   the_sum = the_sum + i
print(the_sum)
```

1.2 Loops - While loops

Alternatively, we can use a `while` loop as follows:

```
the_sum = 0
i = 1
while i <= 100:
    the_sum = the_sum + i
    i = i + 1
print(the_sum)
```

The above is a very common pattern with `while` loops.

- One often has to initialise a variable before the `while` loop begins.
- Then, you can check the condition after each execution of the loop. You'll practice with this technique in the code challenges.

While loops have one big disadvantage: it is not difficult to construct loops that will linger forever. Although many situations call for while loops, our experience is that they are far less common than `for` loops in practice.

1.2 Loops - Nested structures and conditional statements

You may nest loops in Python. Consider the following Python pseudo-code:

```python
for outer_variable in outer_iteratable:
    statement_a
    statement_b

    for inner_variable in inner_iteratable:
        statement_c
        statement_d
        statement_e

    statement_f
    statement_g
```

Let's analyse this pseudo-code, again, indentation is important.

- All statements that are indented one level (four spaces) from the first loop are considered part of the first loop (`statement_a` , `statement_b` , `statement_f` and `statement_g`).
- All statements that are indented one level (four spaces) from the second loop are considered part of the second loop (`statement_c` , `statement_d` and `statement_e`). The second loop are ultimately indented two levels (eight spaces).

1.2 Loops - Nested structures and conditional statements

In this case, Python will begin working on the outer loop.

- It will set the `outer_variable` equal to the first item in `outer_iteratable`.
- Then, it executes `statement_a` and `statement_b`.
- Next, Python does the **entire** inner loop. It sets `inner_variable` equal to the first object in `inner_iteratable` and executes `statement_c`, `statement_d` , and `statement_e`.
- Then it sets `inner_variable` equal to the second object in `inner_iteratable`, if it exists, and executes those same statements.
- Once the inner loop is completed, Python finally executes `statement_f` and `statement_g`. Now, Python has completed the first pass through the outer loop. It sets `outer_variable` to the second object in `outer_iteratable`, if it exists, works through the code again.

1.2 Loops - Nested structures and conditional statements

Example - to go through all months in 2018 through 2020:

```
years = [2018, 2019, 2020]
months = [
    "Jan",
    "Feb",
    "Mar",
    "Apr",
    "May",
    "Jun",
    "Jul",
    "Aug",
    "Sep",
    "Oct",
    "Nov",
    "Dec",
    ]

for year in years:
   for month in months:
       print("Year: {}, Month: {}".format(year, month))
```

1.2 Loops - Nested structures and conditional statements

Here, the *outer loop* is over the variable `year` and the *inner loop* is over the variable `month`. Running this code illustrates the process described above.

1. Python first sets the `year` variable equal to `2018`.
2. Then executes the inner loop, setting the `month` variable to each object in turn from `"Jan"` to `"Dec"`.
3. After this, Python sets the `year` variable to the next object in the outer loop: `2019`.
4. Then executes the month loop again. Note that Python goes through the inner loop once for each item in the outer one.

As an experiment, try flipping the order of the `year` and `month` loops above to see how the execution order changes. If you do so, make sure that the `month` for loop is not indented and the `year` for loop is indented four spaces.

You may nest `for` and `while` loops interchangeably as necessary. So, you may have an outer loop that is a `for` loop with an inner `while` loop, or vice-versa.

Nested loops are very useful. However, deeply nested structures can become hard to manage.

- In such situations, you may want to consider using functions or classes to simplify the logical structure of your code.

1.2 Loops - Additional looping control: continue and break

It is often useful to either skip a particular loop iteration or to exit a loop before the `for` or `while` looping is complete. Python has two statements that enable you to do this: `continue` and `break`.

THE `CONTINUE` STATEMENT

The `continue` statement stops evaluating a particular iteration in a loop and proceeds to the next iteration.

- For example, assume you are running a `for` loop over a variable `i` (e.g. `for i in iterable:`). Using `continue` in the `for` loop will stop evaluating the current `i` and return the `for` loop to the beginning with the next `i` from the `iterable`.

1.2 Loops - The `continue` statement

Concretely, let's say we want to add up all the *even* numbers less than or equal to 100. To determine which integers are even, we'll need to use the **modulo operator**.

- The modulo operator `a % b` returns the remainder of dividing a by b. Thus, `7 % 2 == 1` because `2` goes into `7` three times with a remainder of `1`.
- Hence, `a % 2 == 0` if a is even and `a % 2 == 1` if `a` is odd.

Therefore, a simple way to add up these integers is to use an *if* statement inside a loop:

```
sum_of_evens = 0
for i in range(1,101):
    if i % 2 == 0: # i is even
        sum_of_evens = sum_of_evens + i
print(f'Sum of evens is {sum_of_evens}')
```

1.2 Loops - The `continue` statement

Alternatively, we can use a continue statement:

```python
sum_of_evens = 0
for i in range(1,101):
    print(f'Loop is on {i}')
    if i % 2 == 1:       # i is odd
        continue
    print(f'    summing the value of {i}')
    sum_of_evens = sum_of_evens + i
print(f'Sum of evens is {sum_of_evens}')
```

- Python first sets `i` to the value of `1`. As `1` is odd (`1 % 2 == 1`), the `if` statement will be considered satisfied and Python will execute the `continue` statement. This ends the loop iteration, sending Python back to restart the loop.
- Then, `i` is set to `2`, the next value in the range. As `2` is even (`1 % 2` is not equal to 1), Python will determine that the `if` statement condition is not satisfied and skip over any statements that are part of the `if`. Thus, Python executes the `print` statement and updates the sum. The loop starts a new with `i` set to 3.

1.2 Loops - The `continue` statement

Given that you can avoid evaluating code using `if` statements, you may wonder why Python has a `continue` statement.
Essentially, embedding a lot of code inside `if` statements can make programs hard to read, and `continue` statements can make code more elegant by eliminating deep indentation.
For example, let's say you want to sum up all integers less than 100 that are not multiples of 2, 3, 7, or 13. Using an `if` statement requires Boolean logic:

```python
the_sum = 0
for i in range(1,101):
    if i % 2 != 0 and i % 3 != 0 and i % 7 != 0 and i % 13 !=0:
        print(f'    the_sum is {the_sum}')
        the_sum = the_sum + i
print(f'Sum is {the_sum}')
```

This version requires scanning across the line to see all the bases being excluded.

1.2 Loops - The `continue` statement

Alternatively, with `continue` statements:

```python
the_sum = 0
for i in range(1,101):
    if i % 2 == 0:
        continue
    if i % 3 == 0:
        continue
    if i % 7 == 0:
        continue
    if i % 13 == 0:
        continue
    print(f'    the_sum is {the_sum}')
    the_sum = the_sum + i
print(f'Sum is {the_sum}')
```

While longer, this version is arguably easier to understand. Thus, used correctly, the `continue` statement can help you improve your **code readability** and **maintainability**.

1.2 Loops - The `break` statement

The `break` statement will exit a loop immediately. This can be useful in a number of situations in both `for` and `while` loops.

- For example, consider the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, etc, each number is the sum of the two values immediately preceding it. Thus, if we have currently computed the Fibonacci number 8 (*fib_current = 8*) and we know the preceding number is 5 (*fib_prior = 5*), we then know that the next Fibonacci number is 13. From this point (*fib_current = 13* and *fib_prior = 8*), we can compute the next number and so on.

Let's say we want to find the first Fibonacci number greater than 10,000.

- The following code uses *parallel assignment* to update the current and prior numbers, make sure you can work through how Python will evaluate it to see how it works. We start with a `while` loop implementation:

```python
fib_current = 1
fib_prior = 0
while fib_current < 10000:
    print(f'Fibonacci value is {fib_current}')
    fib_current, fib_prior = fib_current + fib_prior, fib_current
print(f'First Fibonacci value greater than 10,000 is {fib_current}')
```

1.2 Loops - The `break` statement

This is equivalent to the following code using a `break` statement:

```
fib_current = 1
fib_prior = 0
while True:
    print(f'Fibonacci value is {fib_current}')
    fib_current, fib_prior = fib_current + fib_prior, fib_current
    if fib_current >= 10000:
        break
print(f'First Fibonacci value greater than 10,000 is {fib_current}')
```

The `break` statement is not strictly necessary in the code above as we have a simple `while` loop version. However, if there are multiple reasons why you would like to stop executing a loop, having the ability to `break` at any point is very useful.

1.2 Loops - *break* and *continue* in nested loops

When used inside nested loops (i.e. loops within loops), the break and continue statements apply to the active, innermost loop.

- For example, let's say we want to loop over all combinations of positive integers x and y that sum to an even number less than 4. Why do this? Why not?

```python
for x in range(0, 4):
    print(f"Outer loop: x = {x}")
    for y in range(0,4):
        print(f"    Start of Inner loop: y = {y}")
        if x + y >= 4:
            print(f"    x = {x} and y ={y} have sum >= 4, continuing to next y value")
            continue
        elif (x + y) % 2 != 0:
            print(f"    x = {x} and y = {y} have non-even sum, continuing to next y value")
            continue
        print(f"    x = {x} and y = {y} have even sum less than 4")
```

1.2 Loops - *break* and *continue* in nested loops

If, for some reason, we wanted to eliminate the possibility that `y > x`, we could add a break for that condition. When placed in the inner loop, it would stop testing all values where `y > x`:

```
for x in range(0, 4):
    print(f"Outer loop: x = {x}")
    for y in range(0,4):
        print(f"    Start of Inner loop: y = {y}")
        if y > x:
            print(f"    y = {y} > x = {x}, breaking out of inner loop")
            break
        elif x + y >= 4:
            print(f"    x = {x} and y ={y} have sum >= 4, continuing to next y value")
            continue
        elif x + y % 2 != 0:
            print(f"    x = {x} and y = {y} have non-even sum, continuing to next y value")
            continue
        print(f"    x = {x} and y = {y} have even sum less than 4")
```

1.2 Loops - Additional examples

The continue and break statements can be confusing at first. Here we provide a few more examples. As you go though each example, try to visualise what each loop will produce.
In particular, remember that:

- **`continue` means "move to the next iteration in the loop right now"**
- **`break` means "exits *this loop* right now"**

The first example:

```python
# Before pressing Run, answer this question:
#   "What sequence does range(0, 10, 2) produce?"
#   "Which numbers will be printed?"
for even in range(0, 10, 2):
    print(even)
    if even > 2:
        break

### 0, 2, 4
```

Note that `range(0, 10, 2)` will produce a sequence with *every other* integer from `0` to `9`. This is because when evaluating `range(start, end, step)`:

- The sequence will stop at `end - 1`
- The `step` parameter determines the increment: so `1` means every integer, `2` means every other integer, etc...

1.2 Loops - Additional examples

The loop above translates into the following:

- First iteration:
    - `even` starts at 0
    - `print(even) --> 0`
    - `if even > 2 --> False` so `break` will not be executed
- Second iteration:
    - `even` is `0 + 2 = 2`
    - `print(even) --> 2`
    - `if even > 2 --> False` so `break` will not be executed
- Third iteration:
    - `even` is `2 + 2 = 4`
    - `print(even) --> 4`
    - `if even > 2 --> True` so `break` will execute `-->` exit the loop

1.2 Loops - Additional examples

Now, lets nest this loop into another:
The second example:

```python
# Before pressing Run, answer this question:
#   "Which numbers will be printed?"
for odd in range(1, 10, 2):
    for even in range(0, 10, 2):
        if even > 2:
            break
    print(even, odd)

### 4; 1,3,5,7,9
```

Two things to notice here:

- The `print` statement is outside the inner loop. This means the inner loop will finish before the first print statement.
- The inner loop will exit after `even` is `4`, but the outer loop will keep going.

1.2 Loops - Additional examples

What about the following?
The third example:

```
# Before pressing Run, answer this question:
#   "Which numbers will be printed?"
for even in range(0, 10, 2):
    print(even)
    if even > 2:
        continue
```

In this case, all elements of the sequence will be printed. The `if even > 2: continue` statement is inconsequential.

1.2 Loops - Additional examples

However, the following will only print `0` and `2`:

The fourth example:

```
# Before pressing Run, answer this question:
#   "What sequence does range(0, 10, 2) produce?"
#   "Which numbers will be printed?"
for even in range(0, 10, 2):
    if even > 2:
        continue
    print(even)
```

1.2. Quick Summary: pass and continue, and break

- `pass` : do nothing here, generally used as a placeholder.
- `continue` : move to the next iteration in the loop right now
- `break` : exits *this loop* right now