QLSC 612: Introduction to Python

Michelle Wang

May 2 2023

Modified from the 2022 version of the module, which was taught by Jacob Sanz-Robinson (link to the video recording.

What is Python?

- A programming language
- Simple, easy to learn syntax that emphasizes readability. Efficient and intuitive to write in
- Created 30+ years ago, large community. Huge growth since 2012
- Lots of libraries permitting you to use others' code

Python is a *high-level* programming language

- Strong abstraction from the details of the computer or the machine language.
- Don't have to deal with 1/0's or allocating memory and registers.
- Lots of nice built-in data structures.

Python is an *intepreted* language

- If you code in C/C++, you have to compile it, translating your human understandable code to (OS-specific) machine understandable code which the CPU can execute.
- Python code is translated into **bytecode** (low-level set of instructions that can be executed by an interpreter).
- The bytecode is executed on a **virtual machine** (VM) and not CPU.
- Interpreter checks the validity of variable types and operations (as opposed to having to declare them and having them checked on compilation).
- Advantage: given the bytecode and the VM are the same version, the bytecode can be executed on any platform
    - Cost/inconvenience is that it typically takes a bit more time to run.
- See this Medium article for more information

Python is *object-oriented*

- Organized around **objects**. Everything in python (lists, dictionaries, classes, etc.) is an object (chunk of memory containing a value).
- We won't delve into the object oriented philosophy today.

Python is *object-oriented*

- Organized around **objects**. Everything in python (lists, dictionaries, classes, etc.) is an object (chunk of memory containing a value).
- We won't delve into the object oriented philosophy today.

Python has *dynamic semantics*

- Variables are **dynamic objects**: think of them as labels on actual objects
- Can set a variable to an integer, then to a string, etc.
- Assign variables in a way that makes more sense to a human than it does to the computer.

Demo: running a Python script

- A Python file is a text file. By convention it should have the `.py` extension
- In a command line: Open the terminal window and type in the word `python` (or `python3` if you have both versions of Python installed), followed by the path to your script. For example, if your script is called `hello.py`, you would type `python3 hello.py`.
- Alternatively, you could add the shebang line `#!/usr/bin/env python` to the top of the file, make it executable (e.g., `chmod u+x hello.py` in a terminal), and run it as you would run a shell script (e.g., `./hello.py`)

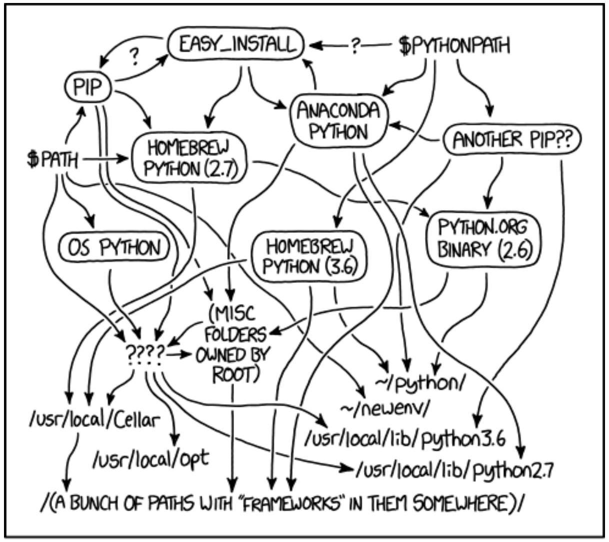
In [1]:

```python
# this code calls the print function
# which prints the specified message to the screen.
print("Hello world!")
```

```
Hello world!
```

Aside: Python virtual environments

- What if you have multiple projects that use different versions of Python or Python libraries?
- It can become very difficult to manage/debug your Python environment



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

https://xkcd.com/1987/

Aside: Python virtual environments

- Solution: create a different environment for each project
  - Each environment has its own dependencies
  - Updating one environment does not after the other ones
- `conda` is a Python package manager that comes with Miniconda/Anaconda
  - Create a new environment named `qlsc612` that uses Python 3.9:

    ```
    conda create --name qlsc612 python=3.9
    ```

  - Activate the environment: `conda activate qls612`
  - Install packages: `conda install [PACKAGE_NAMES]`
  - `conda` cheat sheet
- See also: `venv` as an alternative to `conda`

Demo: using Jupyter Notebooks

- To open the notebook in VSCode, type the following in a **terminal**:

  ```
  # activate the qlsc612 environment
  conda activate qlsc612
  # open the notebook in VS Code
  code <NOTEBOOK_NAME.ipynb>
  ```

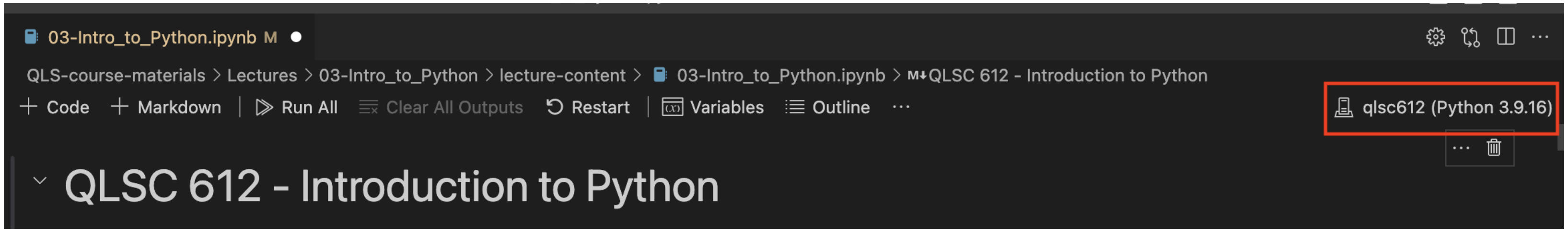
  The notebook should now appear in a new VS Code tab or window.

  If your shell complains that the `code` command cannot be found, you can install it by going to the VS Code Command Palette (`Ctrl`/`Cmd` + `Shift` + `P`) and typing/selecting `Shell Command: Install 'code' command in PATH`.

- Click on a cell to select it, and press `Ctrl` + `Enter` to execute the code.

Demo: using Jupyter Notebooks

When running a Jupyter notebook in VS Code, you may also need to specify the Python environment (kernel). There will be a `Select Kernel` button in the top right corner of the Jupyter notebook, click it and select the one reading something like *qlsc612 (Python 3.9.x) miniconda3/envs/qlsc612/bin/python*. The button should be updated to read *qlsc612 (Python 3.9.x)*. This is the Python environment we have just created for this course: make sure it is the one you are using for later modules.



Aside: Jupyter Notebooks

Jupyter notebooks are interactive documents that can combine text elements and code. They can be handy for exploring a dataset or presenting a report, but they are not necessarily the best choice for running a full-fledged analysis. Running a notebook also requires some external dependencies (e.g., `jupyter`, or `ipykernel` if using VS Code with the Jupyter Notebook extension).

Notebooks are also more complicated than simple Python scripts and modules: refer to the documentation if you are curious about how they work.

In [2]:

```
# the insides of a .ipynb file
! head "03-Intro_to_Python.ipynb" # use '!' for a shell command (head: print 10 first lines
```


```
{
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {
    "slideshow": {
     "slide_type": "slide"
    }
   },
   "source": [
```

Comments in Python

- A comment follows the `#` symbol. Anything following this symbol is not executed.
- Comments help make your code more human-readable. Add comments to clarify what your code is doing.
- It will help other people who see your code, and could even help you when you are re-reading your own code in the future.

In [3]:

```
# This is a comment.
```

Simple data types

- Variables can store data of different types, and these different types can do different things.
- We can use different operators on different data types.
- Python has many built-in data types, in this section we will see some of the widely used, fundamental ones.

Simple data types: `int` (integer)

Positive or negative whole number (no decimal point).

In [4]:

```
print(5)
print(-5) # can be negative
```

```
5
-5
```

Simple data types: `float`

Real numbers. A decimal point divides them into the integer and fractional parts.

In [5]:

```
print(5.0)
print(5.)    # trailing 0 is not necessary
print(-5.0) # can also be negative
```

```
5.0
5.0
-5.0
```

Simple data types: `string`

Sequence of characters.

In [6]:

```
print("hello")
print('12345') # can use single or double quotes
```

```
hello
12345
```

Simple data types: `bool` (boolean)

Represent one of two values: True or False.

In [7]:

```
print(True)
print(False)
```

```
True
False
```

Checking the data type of a variable

You can use the `type()` function to obtain the data type of a variable.

In [8]:

```
print( type(5) )
print( type(5.0) )
```

```
<class 'int'>
<class 'float'>
```

Variables

Variables are containers for storing data values.
Think of them as names attached to a particular object.

In [9]:

```
age = 12 # use '=' to assign a value to a variable
print(age)

age = 24 # update the age variable
print(age)

age = age + 2
print(age)

age += 1 # add 1 to the value of 'age' and assign it back to 'age'
print(age)
```

```
12
24
26
27
```

Operators

Operators can be applied to variables.

- Assignment operator: `=`
- Arithmetic operators: `+`, `-`, `*`, `/`, `//` (integer division), `**` (power), `%` (modulo/remainder)
- Logical operators: `not`, `and`, `or`
- Comparison operators: `==` (equal), `!=` (not equal), `>`, `>=`, `<`, `<=`
- Other: `is`, `in`, etc.

Assignment vs equality operator

They are not the same!

In [10]:

```
a = 5
a = 4 # assign the value 4 to the variable 'a'
print(a)
```

```
4
```

In [11]:

```
a = 5
print(a == 4) # check equality
```

```
False
```

Operator precedence

The order of operators matters! If you are ever unsure about operator precedence, use parentheses!

In [12]:
```
print(a + c / b)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call las
t)
Cell In[12], line 1
----> 1 print(a + c / b)

NameError: name 'c' is not defined
```

In [13]:
```
print((a + c) / b)
```

```
--------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
Cell In[13], line 1
----> 1 print((a + c) / b)

NameError: name 'c' is not defined
```

Operators and variable types

We get a `TypeError` if we use an operator incorrectly

In [14]:

```
print(True > 'abc')
```

```
--------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Cell In[14], line 1
----> 1 print(True > 'abc')

TypeError: '>' not supported between instances of 'bool' and 'str'
```

In [15]:

```
print('qlsc ' + 612)
```

```
--------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Cell In[15], line 1
----> 1 print('qlsc ' + 612)

TypeError: can only concatenate str (not "int") to str
```

Operator overloading

We can use the `+` operator on integers, floats, or strings

In [16]:
```
print(123 + 489)        # integers
print(123. + 489.)      # floats
print('qlsc ' + '612') # strings
```

```
612
612.0
qlsc 612
```

This is because the `+` operator can work differently for different data types (it is *overloaded*)

Typecasting

- Converting a variable from one type to another
- **Implicit**: done automatically (if possible)
- **Explicit**: using a function (e.g., `str`, `int`, `float`)
    - Note that loss of data may occur (for example if casting a `float` to an `int`)

In [17]:
```
print(123 + 489.)          # implicit typecasting (note that the output is a float)
print('qlsc ' + str(612)) # explicit typecasting
```

```
612.0
qlsc 612
```

In [18]:
```
# casting string to int
print(int('612'))
print(int('qlsc')) # invalid
```

```
612
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
Cell In[18], line 3
      1 # casting string to int
      2 print(int('612'))
```

```
----> 3 print(int('qlsc')) # invalid

ValueError: invalid literal for int() with base 10: 'qlsc'
```

Strings

- A sequence of characters in between quotation marks (single or double, either works).
- A single character is a string of length 1.

In [19]:

```
message = "Hello, I am a string"
print(message)
```

```
Hello, I am a string
```

String indexing

- String indexing allows you to access a particular character in a string
- Indexing starts at 0 in Python!

In [20]:

```
message = "Hello, I am a string"

print(message[0])  # first character
print(message[1])  # second character
print(message[-1]) # last character
print(message[-2]) # penultimate character
```

```
H
e
g
n
```

String slicing

- Selecting a substring from a string.
- The first index is where the slice starts (inclusive), second index is where the slice ends (exclusive)

In [21]:

```
message = "Hello, I am a string"

print(message[7:])   # 8th all the way to last character
print(message[7:11]) # 8th to 11th character

print(message[7:-7]) # can use negative indices
```

```
I am a string
I am
I am a
```

Strings are immutable: they cannot be modified

In [22]:
```
message = "Hello, I am a string"
message[1] = "Y"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[22], line 2
      1 message = "Hello, I am a string"
----> 2 message[1] = "Y"

TypeError: 'str' object does not support item assignment
```

In [23]:
```
message = "Hello, I am a string"
message = "Y" + message[1:] # we can make a new string and assign it to the same variable
print(message)
```

```
Yello, I am a string
```

Some string methods/operations on strings

See the docs for more!

In [24]:

```python
message = "This is a string!"
print(message)

print(len(message)) # length of strings

print(message + " And you can add stuff on.") # creates a new string

print("string" in message) # True if "string" is inside the message variable

print(message.count("i")) # Counts the number of times 'i' appears in the string
print(message.find("s"))  # Finds the index of the first 's' it finds in the string
```

```
This is a string!
17
This is a string! And you can add stuff on.
True
3
3
```

Lists

- Store multiple items in a single variable.
- Comma-separated items between square brackets.
- Items in a list need not be of the same type.
- Lists are **ordered**: each item has a position (index).
  - Ex: `[1,2,3]` is not the same as `[3,2,1]`
- Lists are **mutable**: can be changed without entirely recreating the list
  - Elements can be modified, replaced, added, deleted, order changed

In [25]:

```
my_list = [1, 2, 345, 42]
print(my_list)
```

```
[1, 2, 345, 42]
```

Some list operations

In [26]:

```
my_list = [1, 2, 345, 42]

print(my_list[0])      # list indexing (just like strings)
print(my_list[0:3])    # list slicing (just like strings)
print(len(my_list))    # getting the number of items in a list
print(345 in my_list) # checking if an item is in the list
print(sum(my_list))    # computing the sum of the items
```

```
1
[1, 2, 345]
4
True
390
```

Modifying a list (recall that lists are mutable)

In [27]:

```
my_list = [1, 2, 345, 42]

print(my_list.append("hello")) # this does not return anything
print(my_list)                 # my_list is changed
my_list.append([3, "hi", 4])   # appending a list to a list. Now there is a list inside anot
print(my_list)
print(my_list[5][0])           # access the first element ( [0] ) of the list within a list.

my_list[0] = 22                # change the value, lists are mutable
print(my_list)

del my_list[0]                 # deleting an item
print(my_list)
```

```
None
[1, 2, 345, 42, 'hello']
[1, 2, 345, 42, 'hello', [3, 'hi', 4]]
3
[22, 2, 345, 42, 'hello', [3, 'hi', 4]]
[2, 345, 42, 'hello', [3, 'hi', 4]]
```

In [28]:

```
# we can concatenate lists with the + operator (this creates a new list)
list1 = [1,2,3]
list2 = [4,5,6]
list3 = list1 + list2
print(list3)
```

```
[1, 2, 3, 4, 5, 6]
```

Be careful when assigning a list to multiple variables!

In [29]:

```
listA = [0]
listB = listA
listB.append(1)
print(listB) # we changed this
print(listA) # this is also changed (they are the same list)
```

```
[0, 1]
[0, 1]
```

Be careful when assigning a list to multiple variables!

In [29]:
```
listA = [0]
listB = listA
listB.append(1)
print(listB) # we changed this
print(listA) # this is also changed (they are the same list)
```

```
[0, 1]
[0, 1]
```

How can we copy a list?

In [30]:
```
listA = [0]
listB = listA[:]
listB.append(1)
print(listB) # we changed this
print(listA) # this hasn't changed
```

```
[0, 1]
[0]
```

See also: shallow vs deep copies

Tuples

- Similar to lists (**ordered**), but **immutable**: they cannot be changed once they are created
- Declared as a comma-separated list within round brackets.
- Useful for grouping data together. **Allocated more efficiently than lists, and use less memory.**
- Many of the operations and functions we saw for lists also work on tuples (any of them that don't update the tuple).

In [31]:

```
fruit_tuple = ('apple', 'orange', 'banana', 'guanabana')
print(fruit_tuple[3])     # tuple indexing (same as for lists/strings)
fruit_tuple[3] = 'grape' # trying to modify a tuple will cause an error
```

```
guanabana
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Cell In[31], line 3
      1 fruit_tuple = ('apple', 'orange', 'banana', 'guanabana')
      2 print(fruit_tuple[3])     # tuple indexing (same as for lists/strin
gs)
----> 3 fruit_tuple[3] = 'grape' # trying to modify a tuple will cause an
error
```

```
TypeError: 'tuple' object does not support item assignment
```

Typecasting between lists and tuples

In [32]:

```
# we can convert a tuple into a list (and vice-versa)
this_tuple = (1,2,3)
this_list = list(this_tuple)
print(this_list)
```

```
[1, 2, 3]
```

Dictionaries

- Dictionaries store entries as **key:value** pairs. Values can be accessed through their keys.
    - They are an implementation of the *hashmap* data structure
- They are **mutable** and **ordered as of Python 3.7** (before 3.7, they were not ordered).
- Duplicate keys are not allowed
- They are commonly used to store datasets, and to retrieve values from the dataset by specifying the corresponding key.
- To define them, you enclose a comma-separated list of key-value pairs (key and value are separated by a colon) in curly braces.
- Will see the basic functionalty of these data structures, but won't go into too much in depth

In [33]:

```
# keys can be any hashable immutable type, such as strings or integers...or even some tuples
fruits_available = {"apples": 3, "oranges": 9, "bananas": 12, "guanabana": 0}
print(fruits_available["apples"]) # accessing the value associated to the "apples" key
```

```
3
```

Modifying a dictionary

In [34]:

```
# we can store dictionaries inside dictionaries, which are called nested dictionaries.
fruits_nutrition = {"apple": {"calories" : 54, "water_percent" : 86, "fibre_grams" : 2.4},
                    "orange" : {"calories" : 60, "water_percent" : 86, "fibre_grams" : 3.0}}
print(fruits_nutrition["apple"]["calories"]) # note the two key levels

# updating a value in a dictionary
fruits_nutrition["apple"]["calories"] = 52
print(fruits_nutrition["apple"]["calories"])

# adding an item to the dictionary
fruits_nutrition["banana"] = {"calories" : 89, "water_percent" : 75, "fibre_grams" : 2.6}
print(fruits_nutrition) # notice our dictionary now has a new fruit
# this is different for lists, where accessing an inexistent item would cause an error

# deleting an item from the dictionary
del fruits_nutrition['apple']
print(fruits_nutrition.keys()) #list the keys or values
```

```
54
52
{'apple': {'calories': 52, 'water_percent': 86, 'fibre_grams': 2.4}, 'oran
ge': {'calories': 60, 'water_percent': 86, 'fibre_grams': 3.0}, 'banana':
{'calories': 89, 'water_percent': 75, 'fibre_grams': 2.6}}
dict_keys(['orange', 'banana'])
```

Some dictionary methods

See the documentation for more!

In [35]:
```
fruits_nutrition = {"apple": {"calories" : 54, "water_percent" : 86, "fibre_grams" : 2.4},
                    "orange" : {"calories" : 60, "water_percent" : 86, "fibre_grams" : 3.0}}

print(fruits_nutrition.keys()) # list the keys or values
print(fruits_nutrition["apple"].keys()) # for the nested dictionary
print(fruits_nutrition["apple"].values())

print(fruits_nutrition.get("apple"))    # alternative way to obtain a value from a key
print(fruits_nutrition.get("blahblah")) # can test if entry exists without causing an error
print(fruits_nutrition["blahblah"])     # KeyError
```

```
dict_keys(['apple', 'orange'])
dict_keys(['calories', 'water_percent', 'fibre_grams'])
dict_values([54, 86, 2.4])
{'calories': 54, 'water_percent': 86, 'fibre_grams': 2.4}
None
```

```
---------------------------------------------------------------------------
-
KeyError                                  Traceback (most recent call las
t)
Cell In[35], line 10
      8 print(fruits_nutrition.get("apple"))    # alternative way to obtai
n a value from a key
      9 print(fruits_nutrition.get("blahblah")) # can test if entry exists
without causing an error if it doesn't
---> 10 print(fruits_nutrition["blahblah"])     # KeyError
```

```
KeyError: 'blahblah'
```

Sets (very briefly)

- They are **unordered** and **mutable**
- They cannot contain duplicate items
- See the documentation for more information (set operations, set methods, etc.)!

In [36]:

```
list_with_duplicates = [1, 2, 3, 1, 2, 3]
unique_items = list(set(list_with_duplicates)) # cast to set, then back to list
print(list_with_duplicates)
print(unique_items)
```

```
[1, 2, 3, 1, 2, 3]
[1, 2, 3]
```

`if` statements

- An "if statement" is written by using the `if` keyword.
- The code within an `if` statement is only executed if the specified condition evaluates to `True`.
- The code can make decisions based on conditions.
- Can be followed by many `elif` blocks and an `else` block at the end.

In [37]:

```
# x = 7
x = 2
y = 3

if x > y: # note the logic and comparison operators we saw above come in very handy
    print("x is bigger than y")
# else statement provides an alternative plan of action if the condition evaluates to False
else:
    print("x is not bigger than y")

# we can use our logical operators to build more complex conditionals
if x != 7 or y == 3:
    print("Bingo")
```

```
x is not bigger than y
Bingo
```

More `if` statement examples

Logical and comparison operators (and others) can be very useful

In [38]:

```
number_list = [1, 42, 77, 777, 20]

if 42 in number_list:
    print("the number 42 is in the list")

if not 70 in number_list:
    print("70 isn't in the list")

if 42 in number_list and 777 in number_list:
    print("both 42 and 777 are in the list")
```

```
the number 42 is in the list
70 isn't in the list
both 42 and 777 are in the list
```

Loops

- A loop is a sequence of code that is repeated until a certain condition is met.
- There are two main types of loops, `for` loops and `while` loops.
- All `for` loops can be written as `while` loops, and vice-versa. Just use whichever makes your life easier.
- We will see a few examples of how to use them.

`while` loops: execute code for as long as a condition is true

In [39]:

```
i = 1 #initialize our counter
while i < 6:
    print(i)
    i += 1 #note that we are incrementing our counter variable by 1 every time. i += 1 is th
```

```
1
2
3
4
5
```

The `break` statement can terminate a loop early

In [40]:

```
# use the break command to exit the loop - The break statement terminates the loop containin
i = 1
while i < 6:
    print(i)
    if i == 3: # exit the loop when i takes the value of 3
        break
    i += 1
```

```
1
2
3
```

The `break` statement can terminate a loop early

In [40]:

```
# use the break command to exit the loop - The break statement terminates the loop containin
i = 1
while i < 6:
    print(i)
    if i == 3: # exit the loop when i takes the value of 3
        break
    i += 1
```

```
1
2
3
```

Iterating over a list

In [41]:

```
# Using a while loop to iterate over a list
my_list = ["orange", "apples", "bananas"]
x = 0
while x < len(my_list):
    print(my_list[x])
    x += 1 # increment the index
```

```
orange
apples
bananas
```

`for` loops: an easier way to iterate over a sequence (strings/lists/dicts/tuples/etc.)

In [42]:

```
my_list = ["orange", "apples", "bananas"]
for x in my_list:
    print(x)

for y in range(3): # in range(n) - from 0 to n-1, so here it's from from 0 to 2
    print(y)

for y in range(3, 13, 3): # from 3 to 12, in steps of 3
    print(y)

for character in "string": # loop over a string's characters.
    print(character)
```

```
orange
apples
bananas
0
1
2
3
6
9
12
s
t
r
i
n
g
```

Iterating over a dictionary

In [43]:

```
fruits_nutrition = {"apple": {"calories" : 54, "water_percent" : 86, "fibre_grams" : 2.4},
                    "orange" : {"calories" : 60, "water_percent" : 86, "fibre_grams" : 3.0},
                    "banana" : {"calories" : 89, "water_percent" : 75, "fibre_grams" : 2.6}}

for key in fruits_nutrition: # loop over the keys
    print(key)

for item in fruits_nutrition.items(): # loop over keys and values
    print(item)

for key, value in fruits_nutrition.items(): # have acces to both keys and values as you loop
    print(key, "-->", value["calories"])
```

```
apple
orange
banana
('apple', {'calories': 54, 'water_percent': 86, 'fibre_grams': 2.4})
('orange', {'calories': 60, 'water_percent': 86, 'fibre_grams': 3.0})
('banana', {'calories': 89, 'water_percent': 75, 'fibre_grams': 2.6})
apple --> 54
orange --> 60
banana --> 89
```

Nested loops: loops within a loop

Keep in mind that too much nesting can slow down code!

In [44]:

```
my_list = ["orange", "apples", "bananas"]

for item in my_list:
    x = 1
    while x <= 3: # note that the inner 'nested' loop has to finish before the next iteratio
        print(item) # notice the double indentation below.
        x += 1

# Worth mentioning that a break statement inside a nested loop only terminates the inner loo
```

```
orange
orange
orange
apples
apples
apples
bananas
bananas
bananas
```

List comprehension

In [45]:

```
my_list = ["orange", "apples", "bananas"]

new_my_list = [item for item in my_list if item[-1] == 's']
print(new_my_list)
```

```
['apples', 'bananas']
```

Exceptions

- Often when we write code we are faced with handling errors in it. Python offers ways to allow us gracefully deal with these errors.
- Code inside a `try` block lets you test the code for errors.
- The `except` block lets you decide what to do in the case that there is an error inside the `try` block
- The `finally` block allowed you to execute code regardless of the result of the `try` and `except` blocks

In [46]:

```
# in this piece of code, the variable w is not defined, so it throws an error
try:
    print(w) # this code inside the try block is tested for error
except Exception:
    # the code inside the except block is executed if there are errors. The program does not
    print("An exception occurred")
```

```
An exception occurred
```

Using multiple `except` blocks

Go from more specific to more general

In [47]:

```
try:
#     print(int('w')) # TypeError
    print(w) # NameError

# the code throws a name error when it fails outside a try block
# so if we know this is a possibility, we catch it specifically.
except NameError:
    print("Variable w is not defined")

# and this code catches more general errors, in case something else unexpected goes wrong.
except Exception:
    print("Something else went wrong")
```

```
Variable w is not defined
```

Code in the `finally` block is always executed

In [48]:

```
try:
    print(w)
    print("This is not executed")
except NameError:
    print("Something went wrong")
finally:
    print("This executes even if there is an error")
```

```
Something went wrong
This executes even if there is an error
```

Functions

- Block of organized, reusable code that is typically used to perform a single action.
- Can be seen from outside as a black box: **given some input, it returns an output (although they don't always need an output)**.
- While these input names are often used interchangeably, "parameters" refers to the names in the function definition, and "arguments" refers to the values passed in the function call.

In [49]:

```
def summing_two_nums(x, y): # make a new function by using def followed by the function name
    return x + y

print(summing_two_nums(1, 4)) # call the function with inputs 1 and 4.
print(summing_two_nums(3, 3)) # re-use the same function

def appending_to_list(input_list, new_item):
    input_list.append(new_item)
    return input_list

print(appending_to_list([1, 2, 3], 4))
```

```
5
6
[1, 2, 3, 4]
```

Variable scope

Variables defined within a function are known as **local variables**: they stop existing once the function returns.
Variables outside of functions are known as **global variables**. They can be accessed inside functions (provided there is no local variable with the same name).

In [50]:
```
def my_function(my_variable):
    my_variable = 'bar' # local variable with the same name as the global variable
    for i in range(n): # accessing a global variable
        print('my_variable inside the function: ' + my_variable)

my_variable = 'foo'    # global variable
n = 2
print('my_variable outside the function: ' + my_variable)

my_function(my_variable)

print('my_variable outside the function again: ' + my_variable) # unchanged
```

```
my_variable outside the function: foo
my_variable inside the function: bar
my_variable inside the function: bar
my_variable outside the function again: foo
```

Passing mutable objects to functions

- If the function modifies the mutable object, it will also be modified outside of the function.
- This is because the function argument **refers to the same object** as the variable outside the function. It is not a copy.
- Python uses **passing-by-assignment**. See this blog post for more information about passing-by-value vs passing-by-reference vs passing-by-assignment.

In [51]:

```
def change_list(my_list_inside):
    my_list_inside.append([1,2,3,4]);
    print("Values inside the function: ", my_list_inside)
    return # note that we are not returning anything

my_list = [10,20,30];
change_list(my_list);
print("Values outside the function: ", my_list) # changed
```

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

Importing libraries

- Can import libraries to use functions that other people have written and are inside these libraries.
- Some are included by default in Python, and some have to be installed.
- Installing libraries depends on your environment manager
    - `conda install [PACKAGE_NAME]`
    - `pip install [PACKAGE_NAME]`

In [52]:

```python
import math

print(math.pi) # constant
print(math.factorial(5))

from math import factorial
print(factorial(5))
```

```
3.141592653589793
120
120
```

Another `import` example

In [53]:

```
from IPython.display import YouTubeVideo #Importing the YouTubeVideo function from the IPyth
YouTubeVideo("ml6VkmtLXpA",560,315,rel=0)
```

Out[53]:

A very brief introduction to classes and objects

- Classes are used to create **user-defined data structures**. They can have their own variables (called **attributes**) and functions (called **methods**)
- Attributes and methods are accessed with the **dot (`.`) operator**
- An **object** is an instance of a class (class = blueprint) and contains real data

In [54]:

```
class Dog:
    # Class attribute - all objects
    species = "Canis familiaris"
    def __init__(self, name, age): #Specific to instance
        self.name = name
        self.age = age
    def description(self):
        return f"{self.name} is {self.age} years old"

my_dog = Dog("Bonzo", 7)
print(my_dog.name)
print(my_dog.description())
print(my_dog.species)
```

```
Bonzo
Bonzo is 7 years old
Canis familiaris
```

That's it for now!

This was a very brief overview of Python, with the goal of providing you with the basic knowledge needed for the rest of the course. There are plenty of resources online (websites/articles/videos) if you want to learn more on your own. I also recommend the Think Python 2e textbook.

Remember, the only way to learn (and/or become better at) programming is through practice!