Python Introductory Course!

Author: Jamesong7822

Last Updated: 28/12/2018

Data Types

Python has the following few data types:

1. Integers
2. Floats
3. Strings
4. Bools

How do I know what type something is?

```
#This is a comment
#You can print data types using python's inbuilt function 'print' and 'type'
#Try the following code below!
print(type(1)) #Integer
print(type(1.5)) #Float
print(type("two")) #String
print(type("1")) #String
print(type(True)) #Bool
```

Converting between the different data types are easy too! BUT the data must match the requirements of the desired form of conversion as well!

```
#Converting from string to integer
mystring = "1"
myinteger = int(mystring)
print(type(myinteger))

#Converting from integer to string
myinteger = 1
mystring = str(myinteger)
print(type(mystring))

#Converting from float to integer
myfloat = 1.0
myinteger = int(myfloat)
print(type(myinteger))
#BUT BE CAREFUL!
myfloat = 1.1
myinteger = int(myfloat) #Why does this give an erorr?

#Converting from integer to float
myinteger = 1
```

```
myfloat = float(myinteger)
print(type(myfloat))

#TRY the other converts on your own! :D
```

Arithmetic Operations

```
#Declaration of values
a = 2
b = 5
c = 8
d = 4
e = -3

#Addition
print(a+b)

#Subtraction
print(c-b)

#Multiplication
print(a*b)

#Division
print(c/d) #Returns float
#Integer Division
print(c//d) #Returns integer (As the name suggestss duhh)
print(a//b) #This is a clearer example of integer division

#Exponential
print(b**a) #This is equivalent to b to the power of a

#Remainder
print(b%a)

#Absolute
print(abs(e))
```

Variables

Variables can be named with any order of alphabets, but it is often wise to choose sensible names to act as your variable of choice.

Assigning and Equating

Variables are assigned using a single equal sign: **=** , whereas variables are crossed check if they are **EQUAL** (*equal in this case is identity of the components being crossed checked*) using 2 equal signs: **==** .

```python
# Here are some examples of variables
a = "1" # var a is assigned to a string 1
b = "2"
a == "1" # Returns True
a == b # Returns False
type(a) == type(b) # True - why? both are strings!
```

Equating can be used in many ways in python - and are mostly used in functions.

```python
# Here's some trivia of equating
# Did you know that the value of True is given 1 and False is 0?
# Try the following!
a = 1
b = 0
a == True # Returns True
b == False # Returns True
```

Is, In

The above 2 are less commonly used but they can be applied in the following ways, as tests for identity or membership.

Is is essentially like an equality check (ie. a double equal sign: **==**), except it checks for the object in question. Basically, what is happening behind the scenes is python runs a function `id()` for the components being checked. `id()` returns the actual id number of the object, and you can clearly verify yourself that they are different for the example below.

```python
a = [1]
a == [1] # True
a is [1] # False - why? The list stored in var a is not the same OBJECT as the list we are comparing with.
```

In is used to check if an element exists within a collection of elements. The following example shows the simple test.

```python
a = 1
a in (1, 2) # True
a in (2, 3) # False
```

String Methods

Slicing of Strings

```
mystring = "I am a boy!"
print(mystring[:5]) # Slice from start of string till index 4
print(mystring[5:]) # Slice from index 5 till end of string
# What do u think this does?
print(mystring[::-1])
# Skipping in String Slicing
# String slicing can do step slicing as well
# Here's how to do it!
print(mystring[::2]) # This prints every 2 elements
print(mystring[::-2]) # This prints every 2 elements but in reversed order
```

String concatenation

```
string1 = "I am"
string2 = "a boy"
# What do you notice about the difference between the ways of concatenation below?
print(string1 + string2) # Output: "I ama boy"
print(string1,string2) # Output: "I am a boy"

# We can fix the first case by doing this
print(string1 + " " + string2)

# Of course, we can also combine + and commas in string concatenation
string3 = "not a girl."
print(string1, string2 + ", " + string3)
```

String Formatting

This is basically a more advanced technique to combine strings together!

```
mystring = "My name is"
name = "James"
surname = "Ong"
age = 21
print(mystring, name)

# TAKE NOTE THAT '+' can only be used to concatenate data of same type together
# This is why the following line will fail
print("My age is" + " " + age)

# This works, but what if you have a lot of strings to string together (punneddd)
# There are special placeholders you can use
print("My name is %s and my surname is %s" %(name, surname))
# You can even do the same for different data types
print("My name is %s and I am %d years old." %(name, age))

# Personally I prefer this way
print("My name is {}, my surname is {} and I am {} years old.".format(name, surname,
age))
```

Stripping and Splitting

Stripping is the removal of white-space from the string from either ends of the string

Splitting returns a list of the sub-string elements after they have been split into their sub-strings by the given split character

```
mystring = "I am a boy  " # Note the trailing white space
print(mystring.strip())

mystring2 = "a b c d e"
print(mystring2.split()) # This splits the string by the default whitespace, and returns a list

mystring3 = "a,b,c,d,e"
print(mystring3.split(",")) # This splits the string by the given input -in this case ",",                                    returning a list
mystring4 = "a-b-c-d-e"
print(mystring4.spit("-")) # This splits the string by the given input "-", returning a list
```

Upper , Lower Cases, Title

Quite self-explanatory ~

```
# Let's say you want a string to be upper cased
mystring = "james"

# Simply call the method, .upper()
print(mystring.upper()) # Take note that mystring is still 'james'
                        # To change/update mystring as "JAMES" you have to reassign the var
print(mystring)
mystring = mystring.upper()
print(mystring)

# Try out the method .lower() here!
```

```
# Let's say you want only the first letter/alphabet of each word in the string to be upper
# You can employ the method .title() to do so easily!
mystring = "james ong"

# Call the method .title()
mytitlestring = mystring.title()
print(mytitlestring)
```

Replace

Let's say you want to replace character(s) in a string with other character(s)

There's a handy function for that! Introducing '.replace()'

```
# Replace takes in 2 string arguments (x, y). NOTE THAT THEY ARE CASE-SENSITIVE!!
```

```
# x = the string to look out for in your string
# y = the replacement string
# Take not that the original string is not touched/edited and has to be reassigned to a variable if   you want to make use of it

mystring = "jamesong"
mystring.replace("a", "i")
print(mystring) # Notice that the string is still the same?

# Reassignment
mystring2 = mystring.replace("a", "i")
print(mystring2)

# MAX REPLACEMENT
# Replace takes in one more optional input that tells the maximum number of times to replace a       string character by
mystring = "yayyy"
mystring3 = mystring.replace("y", "s", 2)
print(mystring3)
```

Data Collection Types

There are a few handy data collection types in Python. But what does a data collection type mean?

Think of data collections like a container, in which you can store the above mentioned data types.

Lists

Lists are one of the more common data collection types.

```
# How do we instantiate a list?
mylist = [] # This line creates an empty list that you can do stuff to later **KIV**
print(type(mylist))

# Let's say you want to put some elements into a list...
# A quick way to do so is to use python's inbuilt 'list' function
quicklist = list((1,2,3,4,5))
print(quicklist)
print(type(quicklist))
```

List Methods

There are like a ton of list methods -- too many to list (punned). But here are some that are useful:

Retrieving of elements

```
# Lets say you want to RETRIEVE elements from within a list
list1 = [1,2,3,4,5]
# How do you retrieve the 2nd element, in this case (2)?
# Elements within the list are given unique 'index' numbers starting from 0, from left to right.
print(list1[1]) # This line prints the 2nd element in list1, which in our case is 2!
# Try this!
print(list1[-1]) #What do you think this does?
```

List slicing

```
# What about list-slicing?
list1=[1,2,3,4,5]
print(list1[1:]) # From index 1 to the end of the list
print(list1[:5]) # Until index 5 of the list
print(list1[2:4]) # From index 2 to index 3 of the list
```

Append

```
# You can add to a list by using 'append'
list1 = [1,2,3,4,5]
list1.append(6) # This adds 6 to the BACK of the list, ie. the right side of the list
print(list1) # You can easily check it here!
```

Concatenation

```
list1 = [1,2,3]
list2 = [4,5]
print(list1 + list2) # This adds up the 2 lists!
```

Remove

```
# How about removing? You guessed it! Theres a function called 'remove'.
# 'remove' removes the first item from the list (right to left) that has a value of the input
# However, do take care! Errors will be returned if you call 'remove' but the element does not       exist within the list.
list1 = [1,2,3,4,5]
list1.remove(6) # This returns an error, because the element 6 does not exist in list1
list1.remove(5) # This removes the element 5 in the list!
```

Pop

```
# However what happens if you just want to return and remove elements in the list?
# You can use the function 'pop'
# 'pop' by default (with no inputs) will return the last element of the list, as well as remove it   from the list!
list1 = [1,2,3,4,5]
removed = list1.pop() # Here the last element of the list is removed and stored in var 'removed'
print(removed) # Just a double check to see what you have removed!
print(list1) # Again, another check to see what elements still remain in the list!
# Try pop with inputs! how do u think it works?
removed = list1.pop(2)
print(removed)
print(list1)
```

Indexing

```
# 'index' returns the index of the input. IF it exists in the list.
list1 = [1,2,3,4,5]
list1.index(3) # This gives 2
```

Replacing

Since a list is mutable - which means you can freely change the data stored within, one can easily use list as an update log.

```
list1 = [1,2,3,4,5]
# Lets say you want to change the number 3 to a 10, you can do it by:
list1[2] = 10
print(list1)
```

Count

```
# 'count' gives the number of occurences of the element appearing in the list
list1 = [1,2,3,4,5,2,5,2,5,6,5]
print(list1.count(5)) #This gives 3-the number of occurences of the element 5 in the list
```

Sort

```
#'sort' is a useful function to sort elements within the list.
list1 = [1,5,2,7,2,543,72,34,702]
list1.sort()
print(list1)
# Try this!
list2 = [1,5,2,7,2,543,72,34,702]
list2.sort(reverse=True) # What do you think this additional input does?
print(list2) # Verify your guess here!
```

Reverse

```python
# 'reverse' is a (kinda) useful function to reverse the elements within the list
list1 = [1,2,3,4,5,6,7,8,9,10]
list1.reverse()
print(list1)
# Note that this is the same as [::-1]
```

Join

```python
# You can join elements within a list into a string!
list1 = ["james", "ong"]

# What is the difference between the 2?
mystring = "".join(list1) # Here, the 2 strings are joint together
print(mystring)

mystring2 = " ".join(list1) # Here, the 2 strings are joint together but with a whitespace
print(mystring2)

# You can specify any 'delimiter' you want :)
mystring3 = "-".join(list1)
print(mystring3)
```

Whew, that was quite a bunch to remember right? Fret not, with regular use, these methods will be at the tip of your fingertips :D

List Comprehensions

The python list comprehensions are an easy way to apply a function or filter to a list of items.

Here's some examples of using list comprehensions:

```python
squares = [x**2 for x in range(10)]
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# How do you think the following conditional filter works?
uneven_squares = [x**2 for x in range(10) if x % 2]
# [1, 9, 25, 49, 81]

# You can achieve the same result with a for loop as well (below)
uneven_squares = []
for i in range(10):
    if i % 2:
        uneven_squares.append(x**2)

```

Tuples

Tuples are basically just like lists, except that the data you store within tuples are immutable!

Immutable is a property that prevents edits! Thus tuple values are very useful for restricting data edits in your code.

```
# This demonstrates the immutability of tuples
mytuple = (1,2)
mytuple[0] = 3 # An error would be returned as tuples are immutable and do not allow for item                   assignment

```

Tuple Methods

Retrieving of elements

Similar to lists, you can retrieve elements within tuples by calling their indexes

```
mytuple = (1,2,3,4,5)
print(mytuple[2]) # This would print 3, the element at index 2!
```

You can also do the following:

```
mytuple = (1, (2, 3))
a, b = mytuple
# a = 1
# b = (2, 3)
```

This is called unpacking and can also be done for lists.

Concatenation

Just like lists, you can concatenate tuples together as well

```
tuple1 = (1,2,3)
tuple2 = (4,5)
mytuple = tuple1 + tuple2
print(mytuple)
```

Count

Yeah, you can count the number of occurrences as well

```
mytuple = (1,2,3,4,5,2,2,6)
print(mytuple.count(2)) # This prints 3, as there are 3 occurences of 2s.
```

Dictionary

A dictionary is a data collection type that has the structure of key - value. These 2 come in pairs. You use each unique "key" to access the data stored in the corresponding value. Take note that keys are UNIQUE, MUST BE A STRING and cannot be repeated in the dictionary!

If you happen to repeat the keys, only the last occurrence of the key would be taken into consideration!

```
mydict = {"a": A, "a": B}
print(mydict)
print(mydict.values())
```

Values can be any kind of data format

```
mydict = {"string": "howdy!", "list": [1, 2, 3], "tuple": (1, 2, 3, 4), "integer": 58}
# You can access the stuff here
print(mydict["string"])
print(mydict["list"])
print(mydict["tuple"])
print(mydict["integer"])
```

Dictionary Methods

Items

This method returns a tuple containing key and value. ie (key, value).

```
mydict = {"a": A, "b": B, "c": C}
for key, value = mydict.items():
    print("Key: {} has a Value: {}.".format(key, value))
```

Keys

```
# This returns keys within a dictionary
mydict = {"a": A, "b": B, "c": C}
print(mydict.keys())
```

Values

Take note that while keys are UNIQUE within a dictionary, you can afford to have repeating corresponding values

```
# This returns values within the dictionary
mydict = {"a": "hi", "b": "hi", "c": "hi2"}
print(mydict.values())
```

Retrieving of elements

Retrieving elements in dictionaries are easy as 1-2-3, if you know the identifying key to retrieve your values with

```python
# Here's an example of retrieving values!
mydict = {"a": "Good Morning",
          "b": "Good Afternoon",
          "c": "Good Night"
          }

# Let's say you want to get the string "Good Morning"
print(mydict["a"])
```

Dictionary Comprehension

This is similar to list comprehensions. Here's how you set up a dictionary comprehension.

```python
{x: x ** 2 for x in range(10)}
# {0: 0, 1:1, 2:4, 3:9, 4:16, 5:25, 6:36, 7:49, 8:64, 9:81}

# Applying conditional filters as well
{x: x ** 2 for x in range(10) if x % 2}
# {1:1, 3:9, 5:25, 7:49, 9:81}
```

One can also mix the 2 (list n dict), to give a dictionary of list or a list of dictionaries. Try on your own!

The following gives a skeleton of how the above described nested structure would be like:

```python
# Dictionary of lists
dict_of_lists = {1: [1], 2: [1, [2, 3], 4]}
# Try:
{x: x ** 2 [y for y in range(x)] for x in range(5)}

# List of Dictionaries
list_of_dicts = [{1: 1, 2: 2},
{3: 3, 4: 4}]
```

Loops

Loops are basically an iterative operation. What this means is that it allows you to run a block of code a set number of times or an infinite number of times (conditionally of course).

For

For loops are finite loops that repeat themselves for the user specified interval

Format

```
"""
for A in B:
A is the variable reference name
B is the iterable object which you loop through
"""
```

Number Iteration

```
# For instance I would like to print the numbers from 1 to 10
# Instead of writing the print statement 10 times, I can simplify the process by:
for num in range(1, 11): # range is an inbuilt function in python that takes in 2 integer parameters, the starting num and the ending num. Take note that it only iterates up till the number before the ending num - in this case: 10.
    print(num)

# We can do the same for a list also
mylist = [] # Instantiating a list
for num in range(1,11):
    mylist.append(num) # Adds num from each loop to the list
print(mylist)
```

Data Iteration

You can also use for loops to loop through data (Lists, Tuples, Strings, etc.)

```
# Iterating through strings
mystring = "James"
for alphabet in mystring:
    print(alphabet)

# Iterating through Lists
mylist = ["A", "B", "C", "D", "E"]
for element in mylist:
    print(element)

# Iterating through Tuples
mytuple = (1,2,3,4,5)
for element in mytuple:
        print(element)
```

Nested For Loops

For more complex iterations, we can employ nested for loops which are for loops within a for loop, within a for....

```python
# Lets say we want to print the following stuff:
# A1, A2, A3, B1, B2, B3
# Here's how you can do it:
for alphabet in ["A","B"]:
    for num in range(1,4):
        print(alphabet+str(num))

```

While

While loops are conditional infinite loops that run as long as the condition it is checking is True.

Basically, you run while loops for loops you want to run infinitely **WHILE TRUE**, but can be broken out with a conditional variable that can become **False** within the loop.

Never run a while loop without a break or you may have to restart your python IDE

```python
# Dont run this without a condition that you can break out of
run = True
count = 1
while run:
    print(count)
    count += 1 # Increase count's value by 1 with each iteration
    if count > 100:
        run = False # This line will help u break out of the while loop
        # Alternatively you can also use the break keyword
        # break

```

Conditionals

If, Elif, Else

Basically an if-else block is a series of blocks of code that **run only** when conditions are met.

```python
# Here's an example
# Let's say you have a list of strings of variable length, and you only want a list of strings that   have length 6 and below.
given_list = ["hi", "james", "watermelon", "basketball", "torch", "temple"]
keep_list = []
for word in given_list:
        if len(word) <= 6:
            keep_list.append(word)
 print(keep_list)

```

Elif, short for else-if, is used within an if-else block of code, when you have multiple conditions (ie. more than 2 choices/options)

```python
# Here's an example
```

```
# Let's say you have a list of numbers that are integers, and you want to sort them into the         following lists.
num_list = [-5, 5, 3, 8, 10, -9, 12, -17, -4]
less_than_5 = []
more_than_8 = []
others = []

# Firstly you set up a loop
for num in num_list:
    if num > 8: # For num more than 8
        more_than_8.append(num) # Here you append the num that meets the condition > 8 to the                                    corresponding list - more_than_8
    elif num < 5: # For num less than 5
        less_than_5.append(num) # Here you append the num that meets the condition < 5 to the                                  corresponding list - less_than_5

    else: # For any other case
        others.append(num) # For num that dont fit the above specified conditions, they get                                 appended to this list - others
```

Connectors

One can employ multiple conditional checks within if,elif,else. This is handy as it simplifies the num of lines needed.

AND

Recall set-theory in math? AND is basically the same as asking for the **intersection** of 2(or more!) venn-diagrams. It requests for elements that are true for the same conditional checks.

Consider the following case:

```
# Let's say you want to filter out from a list of nums, even and positive integers
num_list = [-2, 4, 15, 9, -38, 93, 28, 0, -1]

# First you set up the iterative for loop
positive_even = []
for num in num_list:
    if num >= 0: # This line filters out nums that are positive
        if num % 2 == 0: # This line checks for nums that are evenly divisible by 2. ie. EVEN NUMS
                positive_even.append(num)

# You can simplify the above code into a single line of conditionals by using the connector "and"
positive_even2 = []
for num in num_list:
    if num >= 0 and num % 2 == 0:
        positive_even2.append(num)
```

OR

Recall set-theory in math? OR is basically the same as asking for the **union** of 2(or more!) venn-diagrams. It requests for the elements that are true for either conditional checks.

Consider the following case:

```
# Let's say you want to filter out from a list of nums, positive or even nums
num_list = [-4, 1, -38, 28, 29, 94, 18, -25]

# Declare/Instantiate a list
positive_or_even = []

# First set up the iterative loop
for num in num_list:
    if num >= 0 or num % 2 == 0:
        positive_or_even.append(num)
```

NOT

Yeah, as the term implies, this basically just reverses the meaning of your statements.

```
mylist = [1, 2, 3]
query = 4
if query not in mylist:
    print ("{} is not in mylist".format(query))

some_bool = False
print (not some_bool) # prints True
```

Functions

One may ask, what is the difference between Functions and Methods? Well, a method is tied down to specific object, and can only be used on that specific object type. For instance, list.sort(), you cannot apply a .sort() method onto a string object. However, you can use the built-in sorted() function to sort both list and string objects

```
# .sort() method is unique for lists!
mylist = [2,4,1]
mystring = "james"

print(mylist.sort())
print(mystring.sort()) # This line will give an error, because a string object does not have the                        method.sort()!

# However, the generalised, and inbuilt function sorted() can be applied on both lists and strings
print(sorted(mylist))
print(sorted(mystring))

```

Defining a Function

So, how does one go about defining a function?

```
# A function requires the following:
# Firstly, you let python know you are defining a function by using the keyword "def"
# Let's say you want to define a function that adds 2 numbers together
# We will call the function add2num
# Secondly, you need to tell the function what inputs to expect - done within ()
# The variable names that you use here are only applicable within the function itself!
# Thirdly, we write the block of code that the function executes
# Ready? here we go!

def add2num(num1, num2):
    # This function adds the 2 numbers together and returns it
    result = num1 + num2

    # Now a function can return nothing or a response
    # Returning a response is very useful for functions, because you can assign a var to take on       the response of the function!
    return result

# Now to test our function!
print(add2num(5, 7)) # This should return 12!
```

You can also assign your function to a variable (*not sure why you would do that if you have given your function a good, sensible name though*).

`a = add2num`

Note that this is not the same as a = add2num()!

One can than apply the above function simply by

```
a = add2num
a(1,2)
# Gives 3
```

Lambda

Python has a lightweight function declaration method. The function declared using `lambda` is anonymous, but you can still assign it to a variable like normal functions.

Here's an example:

```
# Squaring function
lambda x: x*x # Generator Object

# Supports assigning too
a = lambda x: x*x
a(2) # Returns 4
```

Name

A function's name - *what goes after* `def`, can be retrieved using the magic method `.__name__`

Docstring

Docstring is just a helpful description you can include in the declaration of your functions within `"""`. It is usually a one-liner, but feel free to extend if required.

```
def myfunc():
    """ Here is the func description """
    print ("Yo")

print (myfunc.__doc__)
# Here is the func description
```

Built-in-functions

Python has a couple of useful built-in-functions. While there are too many to cover, the following are some of the useful built-in-functions to keep at your finger-tips!

Print

This function is your favourite friend in python! It lets you print useful outputs during code execution blocks, and is used most often for debugging purposes.

To use this function, simply call 'print(*what you want to print*)'

```
# print accepts one input
print("james")
```

Range

The 'range' function takes in at most 3 integer parameters: range(x, y, z), and is commonly used to iteratively generate integer numbers.

x: Starting integer num

y: Ending integer num (*take note that range ends/stops at the number 1 before this specified number*)

z: Step. Basically, the incremental 'step', in which the number is increased. Defaults to 1. You can also input negative values to generate decreasing integer numbers as well!

Consider the following examples!

```python
# Let's say you want to place numbers from 1-10 into a list
list1 = []

for num in range(1, 11):
    list1.append(num)

# Check the list by printing it
print(list1)

# However, what if you want only the even numbers from 1-10?
list2 = []
for num in range(2, 11, 2): # Here, we start from 2 since it is the first even number and set the step value to 2 to generate even numbers
    list2.append(num)

# Check the list by printing it
print(list2)

# Now, for the negative or decreasing number generation example
list3 = []

for num in range(10, 0, -1):
    list3.append(num)

# Check the list by printing it
print(list3)
```

Sum,Max, Min, Mean

```python
# You can sum a collection of numbers by using the function sum()
list1 = [1,2,3,4,5]
summation = sum(list1)
print(summation)

# If you have a list or tuple of numbers, you can easily obtain its max/min values
list1 = [1,2,3,-10,3.5]
print(max(list1))
print(min(list1))


```

Length (len)

Python has a useful length function to calculate the number of elements within the queried item

```
mystring = "james"
print(len(mystring))

mylist = [1,2,3,4,5]
print(len(mylist))

mytuple = (1,2,3,4,5)
print(len(mytuple))

mydict = {"A": 1, "B": 2, "C": 3}
print(len(mydict))
```

Sorted

```
# Sorted is a generalised function that takes in data and sorts them in order. It can take in         either strings, list of numbers or strings, tuples and even dicts, and returns a sorted list.
mystring = "james"
print(sorted(mystring))

mylist = ["james", "barry", "joshua"]
print(sorted(mylist))

mylist2 = [1,4,9,-2,-5]
print(sorted(mylist2))

mytuples = (1,5,2,8)
print(sorted(mytuples)) # Note that the tuple is not changed - it's immutable. Only its elements                          gets sorted and returned as a list
```

Now that you know the basics of the function sorted(), lets take on more advanced sorting tips and tricks!

`sorted()` can take in another parameter, termed "key", which allows for the user to specify a function to sort by.

Consider the following cases!

```
# Let's say you have the following tuple data
data = (("james", 10, "A"), ("mary", 15, "D"), ("john", 8, "C"), ("andrew", 21, "B"))

# Assuming you want to arrange/sort them by age
# What do you notice about age? It is of index 1 for each tuple segment data!
sorted_data = sorted(data, key = lambda x:x[1]) # Check out how lambda works in the lambda section!

# Now what about sorting them by name?
# By default, sorted will sort a given data by the first element!
sorted_data = sorted(data)

# How about sorting by grade?
# There is another parameter you can input: reverse - what do you think it does?
```

```
sorted_data = sorted(data, key = x:x[2]) # reverse is False by default!
sorted_data = sorted(data, key = x:x[2], reverse=True)
```

Map & Zip

`map(function, iterable, ...<additional iterables>...)` returns an iterator (generator) , that applies a function to each item of the iterable, yielding the results. If additional iterable arguments are passed, the function will be applied to the items of the iterables in parallel.

Note: Of course, this means that the iterator will stop when the shortest iterable is exhausted.

`zip(*iterables)` returns an iterator of tuples, where the i^{th} tuple contains the i^{th} element from each of the argument sequences or iterables.

Note: Of course, this means that the iterator will stop when the shortest iterable is exhausted.

```
# NOTE: list is used to print out the generator items.

# Map that adds 2 to each item
list(map(lambda x: x+2, [1, 2, 3]))
# Returns [3, 4, 5]

# Zip that strings together 2 lists to tuples
list(zip([1, 2], [3, 4]))
# Returns [(1, 3), (2, 4)]
```

Filter

`filter(function, iterable)` constructs an iterator from elements of the iterable, for which the function returns `True`. In this case, the function serves as a **sieve** that removes unwanted elements.

```
# Example to remove odd numbers
# AKA only even nums return TRUE
nums = [1, 2, 3, 4, 5, 6]
list(filter(lambda x: not x%2, nums))

# Odd nums can be returned by simply
list(filter(lambda x: x%2, nums)) # Recall that 1 == True, 0 == False :D
```

Yield

Yield is a keyword that is used like return, except that a generator is returned. This allows users to suspend and resume their states between each call to retrieve the result.

Lets take a look at the difference between a generator with a normal function approach.

```
def get_squares(n):
    return [x ** 2 for x in range(n)]
print(get_squares(10))
# Returns [1, 4, 9, 16, 25, 36, 49, 64, 81]

```

```
def get_squares_gen(n):
    for x in range(n):
        yield x ** 2

print(get_squares_gen(10)) # This would give a generator object

# We make use of the keyword next to iterate through the generator object
# Otherwise a for loop works as well~
squares = get_squares_gen(4)
print (next(squares)) # prints: 0
print (next(squares)) # prints: 1
print (next(squares)) # prints: 4
print (next(squares)) # prints: 9
# The following will raise the error StopIteration, in which the generator is exhausted.
print (next(squares))
```

It is highly recommended to use generator functions whenever possible, especially for complicated functions

A reason is that a generator function would be able to start the computation and give you back the first calculation - you don't have to wait for the entirety of a normal function computation.

Secondly, generators take up less memory space - compared to data structures like lists, tuples, etc. However, in terms of *speed*, generators are slower than comprehensions. This is because, generators need to passed through the `list()` constructor.

Did you know: `next` is actually a magic method related to the generator class. Read up more here.

Now, let's say you have an infinite generator. How do you control it to stop at a certain point?

```
stop = False
def counter(start = 0):
    n = start
    while not stop:
        yield n
        n += 1

c = counter()
print(next(c))
print(next(c))
stop = True
print(next(c)) # Raises StopIteration
stop = False
print(next(c))
```

While the above method is viable, we can *employ* the use of a `send` method. It feeds a value to the generator which resumes the execution.

```
# send method for generators
def counter(start=0):
    n = start
    while True:
```

```
        result = yield n
        print(type(result), result)
        if result == "Q":
            break
        n += 1

c = counter()
print(next(c))
print(c.send("Wow!"))
print(next(c))
print(c.send("Q")) # Stops the infinite while loop
```

Tips and Tricks

Argument Unpacking

Let's consider the following function:

```
def myfunc(*args):
    for arg in args:
        print (arg)

# You will get prints for each input you send to the function.
```

The method of unpacking also applies in lists or tuples or dictionaries.

For example,

```
# List
mylist = [1, 2, 3]
a, b, c = mylist # a = 1, b = 2, c = 3
a, *b = mylist a = 1, b = [2, 3]

# Tuple
mytuple = 1,2,3,4 # Note you don't need to specify parantheses for tuple declaration
a, b, c, d = mytuple # a=1, b=2, c=3, d=4
a, *b = mytuple # a = 1, b = [2, 3, 4] # Note that it returns as a list

# Dictionary
mydict = {"1": 1, "2": 2, "3":3}
a, b, c = mydict # a = "1", b = "2", c = "3"
a, *b = mydict # a = "1", b = ["2", "3"]
```

Error Handling

Going back to the add2num function, your function will fail if you "accidentally" input a non number input. How can you make your function more robust and more resistant to such accidents?

Assert

Assert is just a way to tell python what data type you expect an input or a variable to be, or whether the data falls within an expected range of values.

```
def add2num(x, y):
    assert (type(x) == int), "Use numbers!"
    return(x+y)

print(add2num("1", 1)) # This gives an assertion error, with a personalised statement touch to it!
```

Try, Except

One way you can do so, is to employ another built in check using the keywords try and except. It's self explanatory, and is just the same as politely asking python to run a chunk of code, and then providing it with a more useful error print if an error pops up!

```
def add2num(num1, num2):
    # Here's how to use try, except!
    try:
        result = num1 + num2

    except(TypeError): # You can choose to not fill up the error type here also
        # A good way to remind yourself or users, is to tell them where they messed up!
        # By using a simple print statement of course!
        print("Please only use number inputs")
```

Do you guys notice a "bug" in the above code? what happens if you call add2sum with string inputs?

That brings us to the next tip!

Conditional checks (BEST)

```
def add2num(num1, num2):
    if type(num1) == int and type(num2) == int:
        return (x+y)
    else:
        print("Use only number inputs")
```

Decorators

Let's just say decorators are just a *sugary way* to illustrate or encapsulate the idea of chaining functions.

Let's say I have the following:

```
def my_decor(func):
    def wrapper():
        print("Before {}".format(func.__name__)
        func()
        print("After {}"format(func.__name__))

    return wrapper

def myfunc():
    print("Hi")
```

```
11
12 newfunc = my_decor(myfunc)
13 newfunc()
14 # Basically what the above 2 lines are: my_decor(myfunc)()
15 # Before myfunc
16 # Hi
17 # After myfunc
```

In Other Words, decorators wrap a function, modifying its behavior.

`Python` has a simpler way of applying decorators - we make use of the `@` symbol. The following example does the same as the above code.

```
1  def my_decor(func):
2      def wrapper():
3          print("Before {}".format(func.__name__)
4          func()
5          print("After {}"format(func.__name__))
6
7      return wrapper
8
9  @my_decor
10 def myfunc():
11     print("Hi")
12
13 myfunc()
14 # Before myfunc
15 # Hi
16 # After myfunc
```

Note: The inner function `wrapper` (in this case), can be named whatever you want. This generic name is usually used to denote decorator usage.

Decorators can be made to accept arguments as well (*when wrapping more complicated functions of course*). We can do it using `*args` or `**kwargs`. An example has been included below.

```
1  from functools import wraps
2  def do_twice(func):
3      @wraps(func)
4      def wrapper_do_twice(*args, **kwargs):
5          func(*args, **kwargs)
6          func(*args, **kwargs)
7      return wrapper_do_twice
8
9  @do_twice
10 def newfunc(x, y):
11     print (x+y)
12
13 newfunc(1, 2)
14 # 3
15 # 3
```

If you notice, we have `from functools import wraps` and `@wraps(func)` which are new in this code block - fret not! It just allows the function we are gonna decorate (`newfunc`) to retain its original name and docstring [check using `newfunc.__name__` and `newfunc.__doc__`].

Read up more about decorators here.

Class

Python is an object-oriented programming language - everything in python is an object.

> Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of attributes, and code, in the form of functions known as methods. A distinguishing feature of objects is that an object's method can access and often modify the data attributes of the object with which they are associated (objects have a notion of "self"). In OO programming, computer programs are designed by making them out of objects that interact with one another.

`classes` are used in object creation, when objects are created by a class, they inherit the class attributes and methods.

What is a class?

A class is akin to a template that one can employ quickly to create objects with similar properties.

You use the keyword "class" to tell python you are using a class. A class is generally used to instantiate objects that share similar properties (**Attributes**) and have the same type of methods.

Example Class - Friend

Here's an example of how to define a class

```
1  class Friend(): # You can put an inheritable class in the parantheses as well!
2      # This block can be filled with "default" states for variables
3      name = "Default Name"
4      age = "Default Age"
5      height = "Default Height"
6      weight = "Default Weight"
7
8      def __init__(self, name=name, age=age, height=height, weight=weight):
9          # Firstly, you let python know what are the various vars to expect
10         # By attaching a self, you are allowing these variables to be accessed
   throughout the            object
11         # The object now has these attributable properties which can be called out!
12         self.name = name
13         self.age = age
14         self.height = height
15         self.weight = weight
16
17 # Instantiate a friend object, and assign it to the variable friend - which no
   inputs
18 friend = Friend()
19 # This friend object now has the default attributes which can be accessed here
```

```
print(friend.name)
print(friend.age)
print(friend.height)
print(friend.weight)

# Instantiate a friend object, and assign it to the variable friend1
friend1 = Friend("James", 15, 170, 60)

# You can access the various properties of the object you have just instantiated here!
# Name
print(friend1.name)
# Age
print(friend1.age)
# Height
print(friend1.height)
# Weight
print(friend1.weight)
```

Class Methods

Magic Methods

These are special methods we can define within the class, and are enclosed within double underscores (`__<method name>__`).

A common magic method which is used to initialize the object instance is `__init__`. This magic method runs right after the object is created.

You can then call the magic method by `<method name> (<object instance>)`.

Functions / Methods

This are the related functions [known as methods], that are usable by objects we instanced from our class.

We define such methods with the normal function declaration, except they must minimally take in a `self` parameter. For instance `def newmethod(self):`.

Instantiation

This is as easy as `obj = <class name>()`.

Modules

Modules are godsend for python! They are one of the many reasons why python is so popular, as modules can be readily imported *easily* in python to be built upon.

This section will cover on the basics of using some modules. For more in-depth guides of using certain modules, please look up for my other guides!

Importing

Importing modules is as easy as 1-2-3 in python

Simply type the one liner code below!

```
import A # A is the name of the module you intend to import

# If you only require a method B from a module A, you can further specify it by:
from A import B

# Further specification can be done to the reference term used to call the module or method by       using the keyword "as"
import A as C # C is the reference term you use to call module A in your code sections

# OR

from A import B as C # C is the reference term you use to call method B in your code sections
```