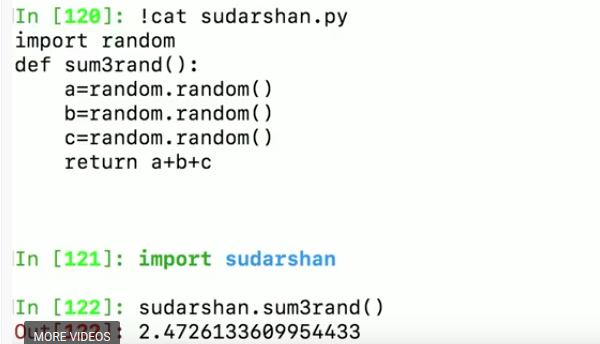
\*\*object binding



Python cheat sheet

## Installation directory

import sys

sys.executable

'c:\\Python26\\python.exe'

Import os

os.

All the variables defined in current namespace can be found by dir()

## COPY

## Copy (Shallow Copy)

If you want to make a copy of dev\_list\_1, use the 'copy' function available in Python. In order to use the copy function, you must import it from the Python standard library which is done using the import statement.

from copy import copy

dev\_list\_2 = copy(dev\_list\_1)

In this example, you are actually copying the entire list, and all items within it. Using the copy function gives dev\_list\_2 a completely new copy of dev\_list\_1. After a copy, changes to dev\_list\_2 have no effect on dev\_list\_1, and vice versa.

Note that the copy function is actually doing a 'shallow' copy. A shallow copy, in computer programming, means that the only first level of items is copied. If you happened to have a list that had complex data structures such as a list of lists, or a list of dictionaries, a shallow copy does not make a copy of those referenced items—it only copies the first-level items. If you wish to copy all items, no matter how deeply nested are the data structures, you must do a deep copy.

## Copy (Deep Copy)

If you need to do a deep copy on your list, use the deepcopy function in Python:

from copy import deepcopy

complex\_list\_2 = deepcopy(complex\_list\_1)

The result is that all data from complex\_list\_1 is copied to complex\_list\_2.

### For loop:

d1= {x:x\*\*2 for x in range (11)}





## String Manipulation

## Binary search

Regular Expression:

vs = 'Cisco IOS XR Software, Version 21.34.685.9'

vp = re.compile('Version ([0-9])\*\.([0-9])\*\.([0-9])\*([0-9])\*')

vm = vp.search(vs)

v = vm.group(1)

\*\*\*\*\*\*\*\*\*\*\*Check more about this \*\*\*\*\*\*\*\*\*\*\*\*\*\*

### Range:

range(i,j,k) increments by k; i,i+k,…,i+nk

Stops with n such that i+nk < j <= i+(n+1)k

Count down? Make k negative!

range(i,j,-1), i > j, produces i,i-1,…,j+1

In python 2: range is a list, where as in python 3 , its not.

Convert by:

kl=list(range(1,10))

In [23]: type(str(70))

Out[23]: str

### String & list

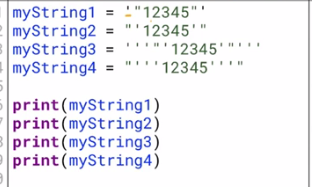
Every print output is string, implicitely converted by print function

]: "Eth1".startswith("et")

Out[7]: False

To include “” and ‘ , put everything in “””….”””

Simillarly “”” can be used when you want to assign multiple line paragraph to a vraible



In [39]: k.extend([2,3])

In [40]: k

Out[40]: [1, 2, 3, 4, 5, 0, 2, 3]

In [41]: k.append(0,4)

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-41-11db76f65752> in <module>()

----> 1 k.append(0,4)

TypeError: append() takes exactly one argument (2 given)

In [55]: 5 in k

Out[55]: True

In [56]: 50 in k

Out[56]: False

### Slice

s = "hello"

s[1:4] is “ell"

s[i:j] starts at s[i] and ends at s[j-1]

Cannot update a string “in place” -> immutable

s = "hello", want to change to "help!"

s[3] = "p" — error!

Instead, use slices and concatenation

s = s[0:3] + "p!"

While slicing, o/p is also a string.

In list, which is mutable, slice gives the value not a sub list

nested = [[2,[37]],4,["hello"]]

nested[0] is [2,[37]]

nested[1] is 4

nested[2][0][3] is "l"

nested[0][1:2] is [[37]]

Unlike strings, lists can be updated in place

nested = [[2,[37]],4,["hello"]]

nested[1] = 7

For immutable values, we can assume that

assignment makes a fresh copy of a value Values of type int, float, bool, str are immutable

Updating one value does not affect the copy

Omitting both end points gives a full slice

l[:] == l[0:len(l)]

To make a copy of a list use a full slice or concatenation

list2 = list1[:]

### Compare

== check value equality

‘is’ check same object/same memory location

list1 = [1,3,5,7]

list2 = [1,3,5,7]

list3 = list2

x == y checks if x and y have same value

x is y checks if x and y refer to same object

list1 == list2 is True

list2 == list3 is True

list2 is list3 is True

list1 is list2 is False

### Concatenation

Note that + always produces a new list

list1 = [1,3,5,7]

list2 = list1

list1 = list1 + [9]

list1 and list2 no longer point to the same object

## Conditions

### IF

The inline form is:

x = y if condition else z

Numeric value 0 is treated as False

Empty sequence "", [] is treated as False

Everything else is True

if … elif … else — conditional execution

for i in … — repeat a fixed number of times

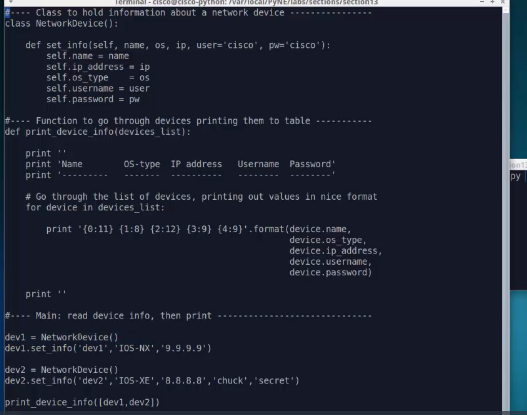
while … — repeat based on a condition

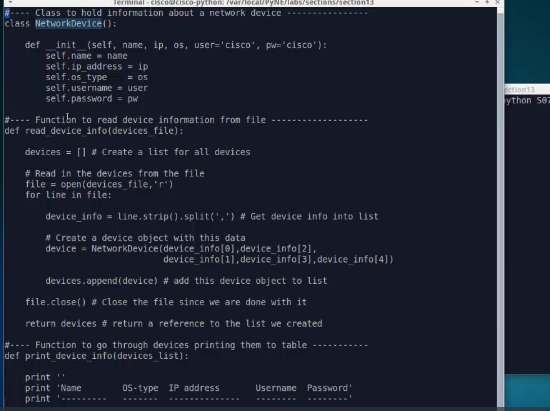
#### continue: Skip the rest of the current iteration of the closest for loop and **continue** with the next iteration of the loop (if there is a next iteration).

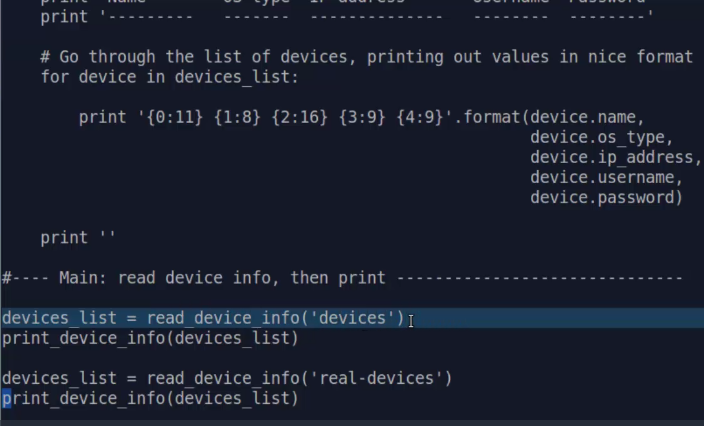
#### break: Skip the rest of the current iteration of the loop and **break** out of the loop altogether, skipping any later iterations, too.

#### pass: Designate an 'empty' body for a control structure.

## Fuction



2. 



### Fn vs methods

The functions we've defined have been

defined at the top level of our programs.

We've created them before actually

writing the body of our programs,

and that's why they're visible.

Methods, on the other hand, are contained within data types.

Instead of just calling them directly like functions,

we have to first say which variable we're referring to

and then call the method inside that variable.

So a method is a function contained

inside a data structure.

End of transcript. Skip to the start.

*Side effect*

When list updated within the scope of a function, its globally changes, where as an int , its scope changes only locally.

def update(l,i,v):

if i >= 0 and i < len(l):

l[i] = v

return(True)

else:

v = v+1

return(False)

ns = [3,11,12]

z = 8

update(ns,2,z)

update(ns,4,z)

ns is [3,11,8]

z remains 8

A function can call itself — recursion

>>> checki(8)

False

>>> def checki(y):

... return(y is 5)

...

>>> checki(5)

True

### SCOPE:

Money = 2000

def AddMoney():

# Uncomment the following line to fix the code:

# global Money

Money = Money + 1

print Money

AddMoney()

print Money

or return(Money + 1)

To answer the question in your subject line,\* yes, there are closures in Python, except they only apply inside a function, and also (in Python 2.x) they are read-only; you can't re-bind the name to a different object (though if the object is mutable, you can modify its contents). In Python 3.x, you can use the [nonlocal](https://docs.python.org/3/reference/simple_stmts.html?highlight=nonlocal#nonlocal) keyword to modify a closure variable.

def incrementer():

counter = 0

def increment():

nonlocal counter

counter += 1

return counter

return increment

increment = incrementer()

A closure binds values in the enclosing environment to names in the local environment. The local environment can then use the bound value, and even reassign that name to something else, but it can't modify the binding in the enclosing environment.

### Conditions together

P=1

Q=2

R=3

Result= (P+Q) > R

#Return false

## DATA structures

* **Lists:** sequences of items, which are indexed by number. Items can be heterogeneous (of different types). Items can be added to a list, removed from a list, and inserted into a list.
* **Dictionaries:** unordered tables of key-value pairs, which are indexed by the value of the key. Items can be added and removed from a dictionary.
* **Tuples:** sequences of items like lists, but a tuple is immutable, meaning its items cannot be changed.
* **Sets:** unordered collections of items. Items can be added or removed from a set, but there can be no duplicate values.
* Converting a tuple to a list:
* device\_tuple = ('10.3.21.5', 'username', 'password')
* device\_list = list(device\_tuple)
* Converting a string to a list:
* device\_string = '10.3.21.5 ,username, password'
* device\_list = device\_string.split(',')
* List comprehensions are a short way of creating a list, using syntax that is not regularly found in other languages.
* Lists are typically created using normal for loops or nested for loops. Consider an unstructured set of data consisting of a device [IP address](https://ondemandelearning.cisco.com/cisco-ems/prne/search?query=IP%20address), username, and a password:
* device\_string = ' 1.1.1.1, username , password '
* To create a list that does not include the extra white space above, you could do the following:
* info\_list = list() # create empty list
* device\_info = device\_string.split(,) # create list from device\_string
* for item in device\_info:
* item = item.strip() # strip white space from device
* info\_list.append(item) #add to info\_list
* To accomplish the same task using a list comprehension:
* device\_string = ' 1.1.1.1, username , password '
* info\_list = [ device.strip() for device in device\_string.split(',') ]

### **Dictionary**

**Assignment:** Assigning one variable name to reference (point to) another object that always exists. In the example, dev1 and dev2 will reference the same object:

dev2 = dev1

* **Copy:** You can make a copy – a shallow copy – of a dictionary by using the 'copy' function. Unlike lists, you do not need to import the copy function to do a shallow copy of a dictionary.

dev2 = dev1.copy()

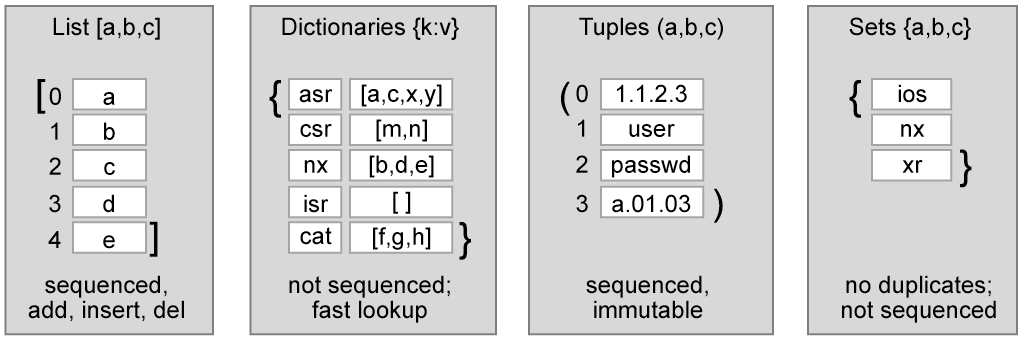
* Deep copy: Shallow copies only make copies of the first level of items in a data structure. Doing a deep copy – that copies all levels of items – requires importing and using the deepcopy function.
* from copy import deepcopy

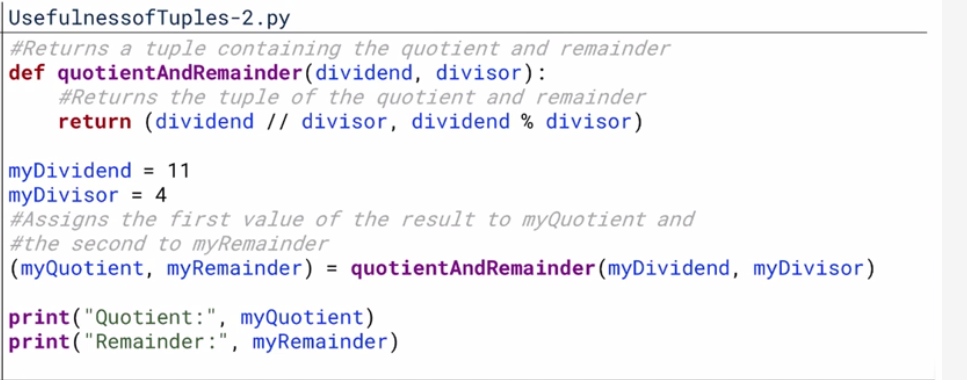
dev2 = deepcopy(dev1)

### Tuple

## Comparing Tuples and Lists

* Tuples are more efficient than lists
* Tuples cannot be accidentally changed
* Tuples are actually used by Python for passing parameters to functions
* Named tuples are sometimes a better way to use tuple data structures, allowing you to refer to the items in a tuple by their name, rather than by the numerical index.
* Since tuples never change, if they consist of immutable objects like strings and numbers, they can be used as dictionary keys.





dev\_info = ('iosxrv1', 'A.01.01')

version = dev\_info[1]

Contrast the example with using a named tuple. The setup takes some extra code, but referencing the field within dev\_info is much cleaner:

from collections import namedtuple

Info = namedtuple('Info','name version')

dev\_info = Info(name='iosxrv1', version='A.01.01')

version = dev\_info.version

To use a named tuple, you first declare that you are using named tuple by importing it. Then you create an object class for your device information, in which you define the fields (name and versions).

Creating the tuple itself involves declaring your named tuple class 'Info' and specifying the values for the named fields, 'name' and 'version', which are set to 'iosxrv1' and 'A.01.01' respectively.

At this point, your tuple is specified using the actual name for the field, which is the version.

## Starred expression

a, b = 1, 2 # simple sequence assignment

a, b = ['green', 'blue'] # list asqignment

a, b = 'XY' # string assignment

a, b = range(1,5,2) # any iterable will do

# nested sequence assignment

(a,b), c = "XY", "Z" # a = 'X', b = 'Y', c = 'Z'

(a,b), c = "XYZ" # ERROR -- too many values to unpack

(a,b), c = "XY" # ERROR -- need more than 1 value to unpack

he single star \* unpacks the sequence/collection into positional arguments, so you can do this:

def sum(a, b):

return a + b

values = (1, 2)

s = sum(\*values)

This will unpack the tuple so that it actually executes as:

s = sum(1, 2)

The double star \*\* does the same, only using a dictionary and thus named arguments:

values = { 'a': 1, 'b': 2 }

s = sum(\*\*values)

You can also combine:

def sum(a, b, c, d):

return a + b + c + d

values1 = (1, 2)

values2 = { 'c': 10, 'd': 15 }

s = sum(\*values1, \*\*values2)

will execute as:

s = sum(1, 2, c=10, d=15)

def sum(\*values):

s = 0

for v in values:

s = s + v

return s

s = sum(1, 2, 3, 4, 5)

or with \*\*:

def get\_a(\*\*values):

return values['a']

s = get\_a(a=1, b=2) # returns 1

this can allow you to specify a large number of optional parameters without having to declare them.

And again, you can combine:

def sum(\*values, \*\*options):

s = 0

for i in values:

s = s + i

if "neg" in options:

if options["neg"]:

s = -s

return s

s = sum(1, 2, 3, 4, 5) # returns 15

s = sum(1, 2, 3, 4, 5, neg=True) # returns -15

s = sum(1, 2, 3, 4, 5, neg=False) # returns 15

## Map,Lambda and Zip

a = [1, 2, 3, 4, 5]

b = [2, 2, 9, 0, 9]

zip(a, b)

lambda pair: max(pair) #lambda <input>: <expression>

prrr=list(map(lambda pair: max(pair), zip(a, b))) # map(some\_function, some\_iterable)

## Debug:

we're going to discuss three general kinds of debugging like this, print debugging (print after each line), scope debugging, and rubber duck debugging.

## Print

To avoid space/new line :

Sep=””

print(todays\_date.day,'/', todays\_date.month,sep="")

## Mathematical Opertaions

* Addition, represented by a plus sign (+). For example, 7 + 4 = 11.
* Subtraction, represented by a minus sign (-). For example, 7 - 4 = 3.
* Multiplication, represented by an asterisk (\*). For example, 7 \* 4 = 28.
* Division, represented by a forward-slash (/). For example, 7 / 4 = 1.75.
* Modulus or remainder, represented by a percent sign (%). For example, 7 % 4 = 3 (because the remainder of 7 over 4 is 3).
* Floor division, represented by a double-forward-slash (//). For example, 7 // 4 = 1 (because 7 over 4 is 1.75, which is rounded down to 1).
* Exponentiation, represented by a double-asterisk (\*\*). For example, 7 \*\* 4 = 2401.

## Date

from datetime import date

import datetime

todays\_date = date.today()

current\_time = datetime.datetime.now()

#Don't modify the code above!

#Complete the line below to print today's date with the

#form year/month/date. For example, January 15th, 2017

#would be 2016/1/15.

#print()

print(todays\_date.year,'/', todays\_date.month,'/',todays\_date.day,sep="")

#Complete the line below to print the current time with

#the form hour:minute:second, such as 12:57:15. Don't worry

#about the leading 0s for single-digit times. If it's

#1:05PM and 7 seconds, the correct answer would be:

#13:5:7 (13 because Python uses 24-hour timeby default).

#print()

now = datetime.datetime.now()

print(now.hour,':',now.minute,':',now.second,sep="")

## Input

Defult input value is string

Python evaluates and first, then or.

## File read and write

Reading from a file is simple and getting the actual contents can be done in multiple ways:

* **Open.** The file must first be opened.

my\_input\_file = open("/home/Documents/input.txt", 'r')

* **Read.** There are multiple options for reading the file:
  + readlines(). Allows you to read all lines in the file into your read buffer.

my\_input = my\_input\_file.readlines()

* + readline(). Allows you to read one line at a time into your read buffer.

my\_input = my\_input\_file.readline()

* + read(). Allows you to read a specific number of characters into your read buffer.

my\_input = my\_input\_file.read(10)

If you choose to not bother with specific read operations, and want to iterate over the entire file one text line at a time, you can use a basic for loop. In exploring Python, you will learn about control structures, such as for loops. For now, you will need to know how to read through a file, line by line, using a for loop. The code to read a file is:

my\_file = open('myfile', 'rt')

for line in my\_file:

# code block to handle text data in 'line'

This code opens the file, then immediately jumps into a for loop, which will iterate over all the lines in the file, one at a time. At each iteration, the line of the text file is placed into the variable that is called line, to be used in whatever functionality the code block is implementing.

file = open('sample.txt', 'r')

x = file.read(2)

y = file.read(2)

z = file.read(2)

print y

The value assigned to x will be AB. The value assigned to y will be CD. The value assigned to z will be EF.

* **rite.** There are also options for writing to a file:
  + write(data). To write all data at once.
  + my\_string = "Cisco Systems"
  + my\_output\_file = open("/home/documents/text.txt", 'w')
  + my\_output\_file.write(my\_string)

my\_output\_file.close()

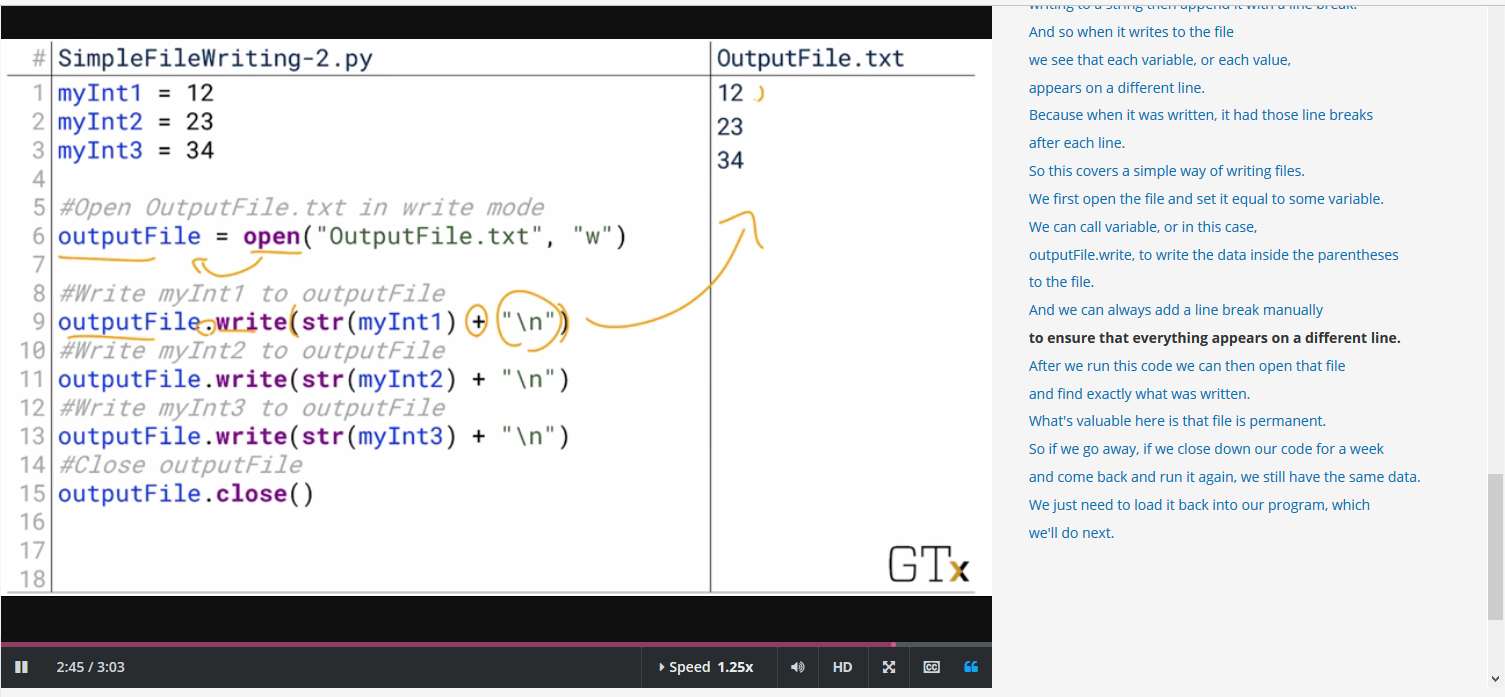
* + writelines(data). To write a few lines of data.
  + my\_string = "This is line 1"
  + my\_output\_file = open("/home/documents/text.txt", 'w')
  + my\_output\_file.writelines(my\_string)

my\_output\_file.close()

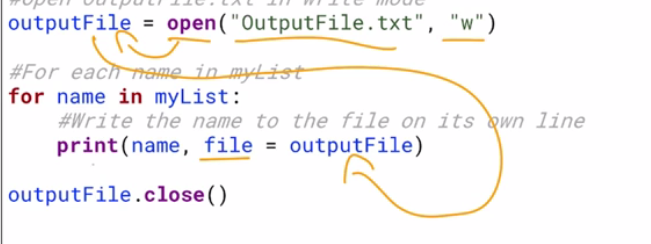
It is also possible to write things out to a file using the 'print' function:

print(data, filename)

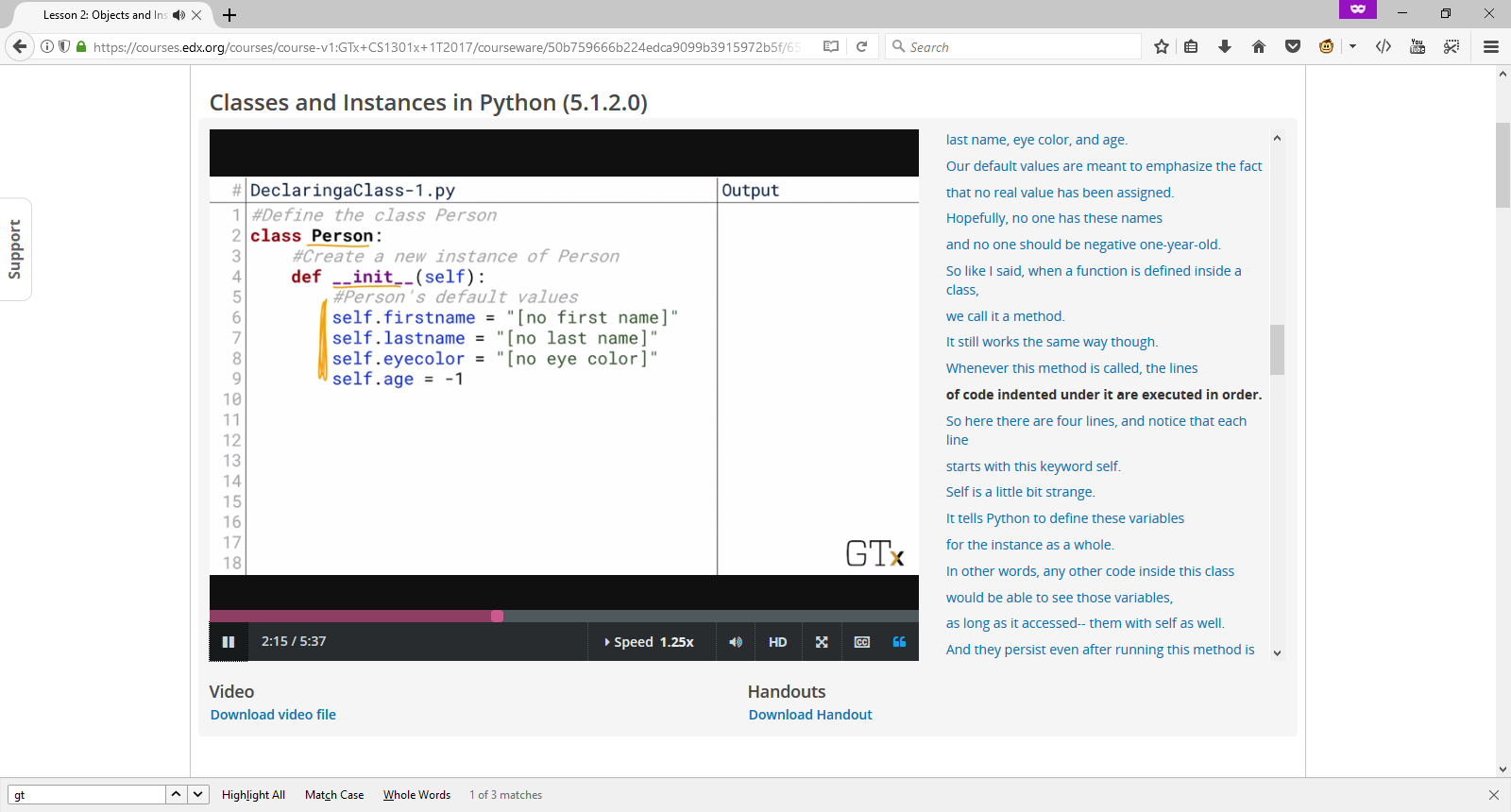
We have to convert the data that we're writing to a string first because the write method can only write strings to files.It only deals with plain text.



to print to file instead of console:



## Class:



Init method is defult run when a class is called and an instance get created. Assigned values are the default values for the variables inside.

notice that each line starts with this keyword self. Self is a little bit strange.It tells Python to define these variables for the instance as a whole. In other words, any other code inside this class would be able to see those variables, as long as it accessed-- them with self as well. And they persist even after running this method is done.

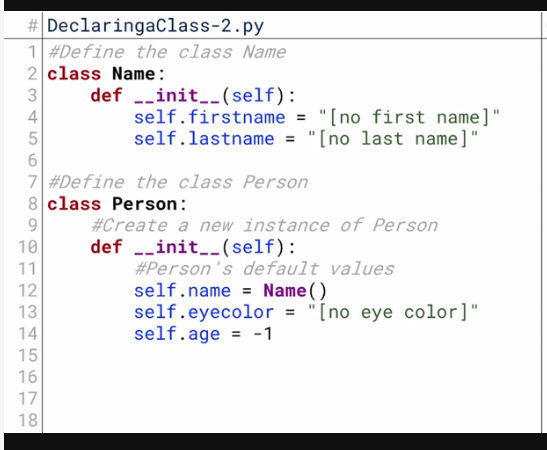
And they persist even after running this method is done.

If we left off self, then the variable age

would cease to exist at the end of this method.

But because it starts with self, the variable age

exists, as long as this class or this instance exists.



Class name() is used in class person for self.name

  
**Constructor:** A common type of method in writing classes that specifies some code to run whenever a new instance of the class is created. The constructor often has parameters that provide values to initialize the variables defined by the class.

**Destructor:** A common type of method in writing classes that specifies how the instance of a class is to be destroyed, such as releasing its memory back to the computer.

**Getter:** A common type of method in writing classes that returns the value of a variable contained within the class. They are commonly used to allow other processing to occur whenever the variable is accessed, like logging.

**Setter:** A common type of method in writing classes that sets a variable contained within the class to a new value. They are commonly used   
to allow other processing to occur whenever the variable is changed, like logging.

### Variable initialization



Or

we've used in the past, in defining our init method,

we could set them equal to some default values.

This means if we create a new person without specifying

any arguments, then it defaults to those default arguments.



By convention, we often perceive variables or methods

that we don't want other classes or functions to access

with a double underscore.

So proceeding in it with a double

underscore as a way of saying don't try

to run this method directly.

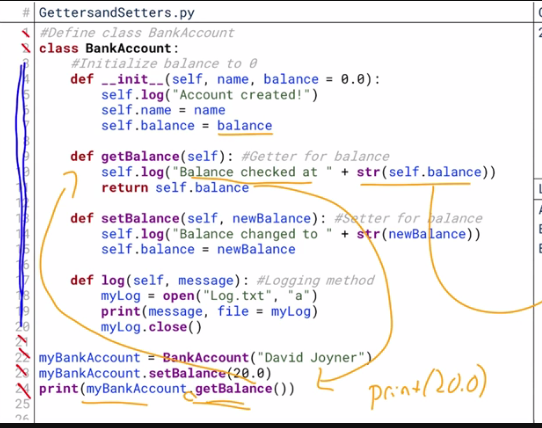
You can, but you shouldn't.

The double underscore simply informs other areas of the code

that those methods or variables are not

meant to be used to access data that way.

### Log fn eg;



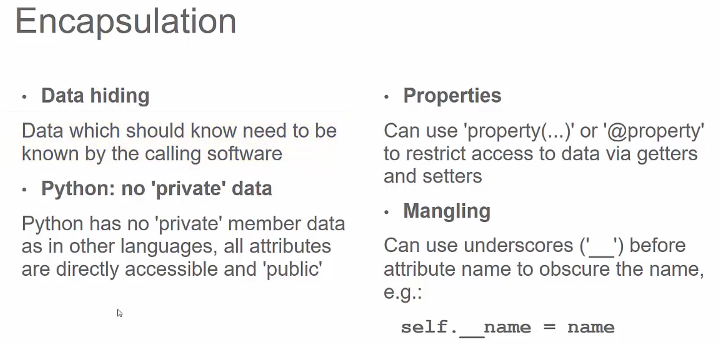
Composition**:**

inheritance represents an is-a relationship. An IOS-XR router is a network device. Composition, on the other hand, represents a has-a relationship. An IOS-XR router has a set of interfaces. An IOS-XR router has a set of static routes.

These examples are fairly straightforward. Some examples that may require careful thought might be the question if an XR router has an [OSPF](https://ondemandelearning.cisco.com/cisco-ems/prne/search?query=OSPF) state? Or is it an OSPF router?

Helping to understand the question may be whether your device would have to inherit from multiple parent classes, for example, if a router is both an OSPF router and an [IPv4](https://ondemandelearning.cisco.com/cisco-ems/prne/search?query=IPv4) router. Inheriting from multiple parent classes is allowed in Python, but quite often it can lead to confusing and complicated code, and thus should be considered with care.

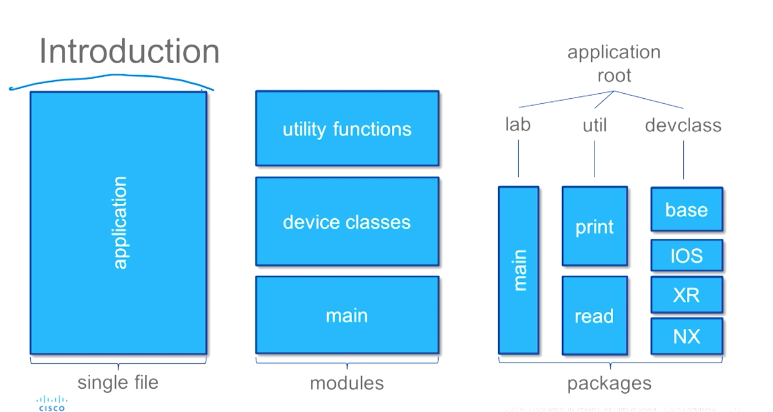
### Encapsulation:



In Python, there are no 'private' attributes or members. Every attribute of the object you have created is visible and accessible to the outside world. If this is a problem, Python provides workarounds to help achieve the hiding of your data:

* **Properties:** Python provides the concept of making your data into a 'property', which makes it inaccessible to outside code. There are two methods of doing this:
  + property(..) function: Using the property function, it is possible to declare certain attributes as only accessible via getter and setter methods, which allow outside code to access the attribute via methods that are implemented in the object – insuring that the attributes are protected from invalid values.
  + @property decorator: Using the property decorator, it is possible to declare attributes as only accessible via a getter function by the same name. The decorator also allows for setting of the attribute.
* **Mangling:** It is also possible to 'mangle' the name of your attributes, by prefacing the attribute name with double underscores. Mangling tells Python that the attribute is intended to be private, and not directly accessible by outside code.

## Modules and packages



### Locating Modules

sys.modules

When you import a module, the Python interpreter searches for the module in the following sequences −

* The current directory.
* If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
* If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python
* The syntax of PYTHONPATH is the same as that of the shell variable PATH.
* Here is a typical PYTHONPATH from a Windows system −
* set PYTHONPATH = c:\python20\lib;
* And here is a typical PYTHONPATH from a UNIX system −
* set PYTHONPATH = /usr/local/lib/python

### The dir( ) Function

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example −

#!/usr/bin/python

# Import built-in module math

import math

content = dir(math)

print content

When the above code is executed, it produces the following result −

['\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', 'acos', 'asin', 'atan',

'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',

'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',

'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',

'sqrt', 'tan', 'tanh']

Here, the special string variable *\_\_name\_\_* is the module's name, and *\_\_file\_\_* is the filename from which the module was loaded.

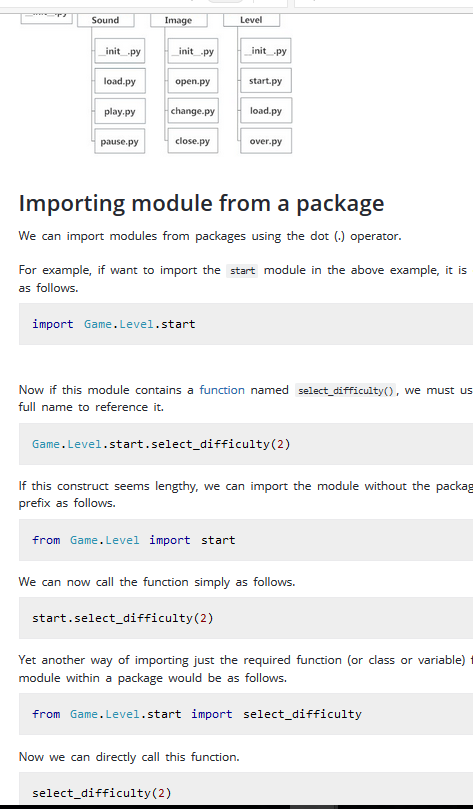
All other names that begin with an underscore are default Python attributes associated with the module (we did not define them ourself).

For example, the \_\_name\_\_ attribute contains the name of the module.

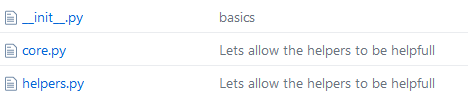
>>> import example

>>> example.\_\_name\_\_

'example'



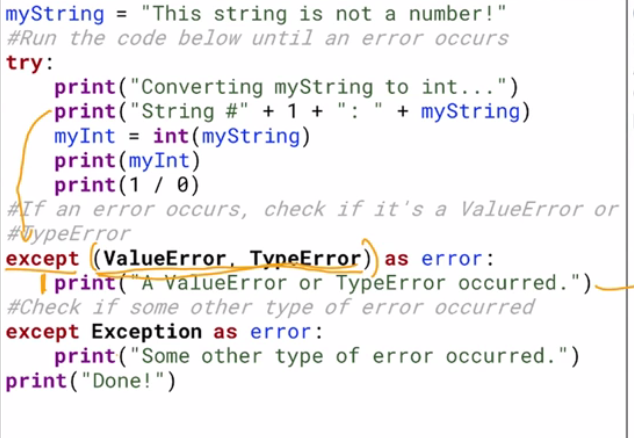
### \_\_init.py\_\_ practices



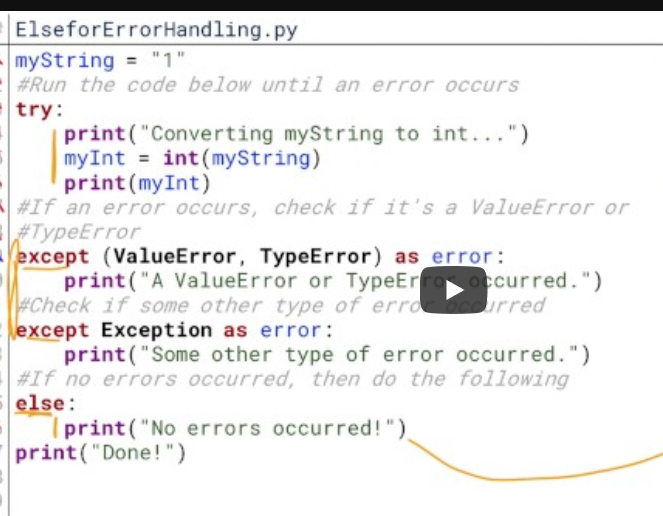
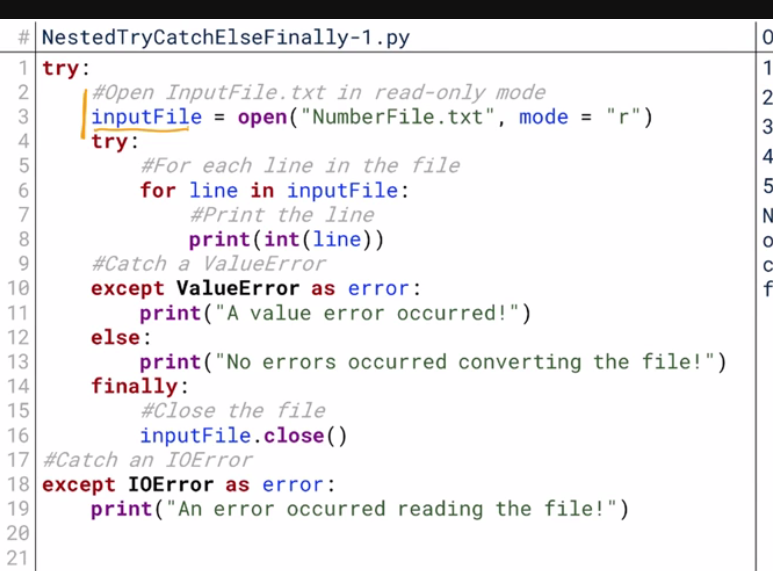
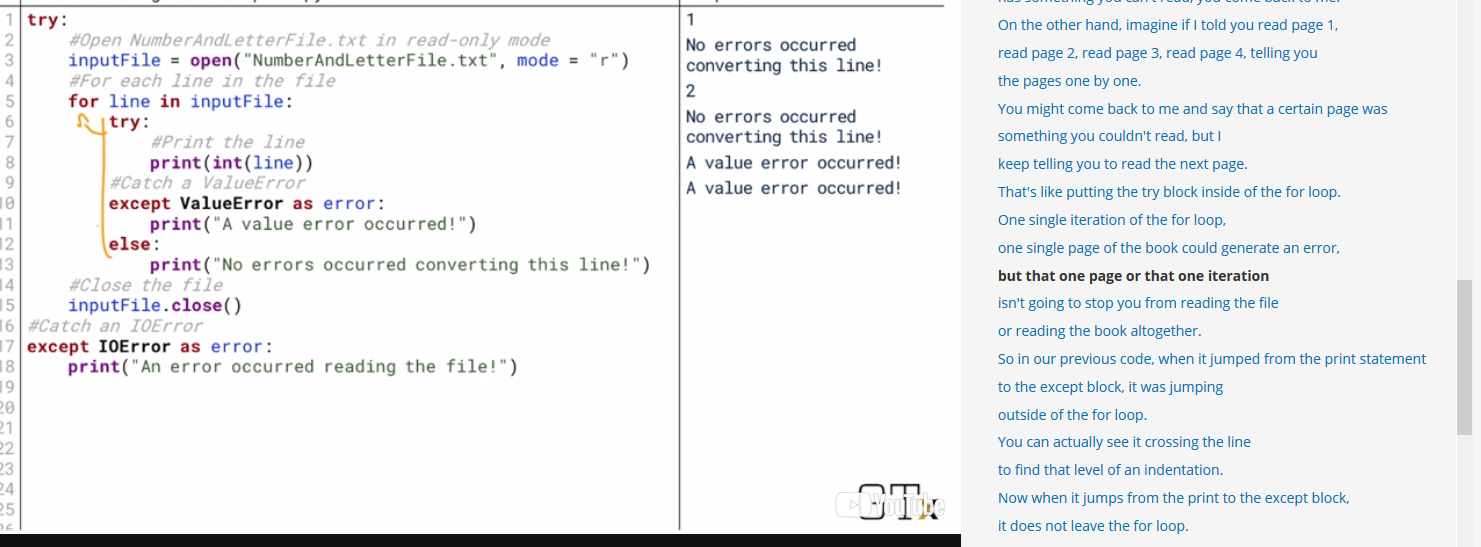
For files in same folder, ie in core.py to access helpers :

from . import helpers

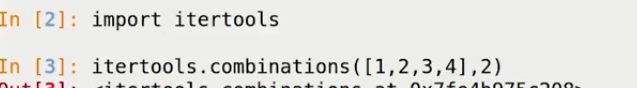
## Error handling



If Two errors occur inside the try block, but the code only runs until it encounters the first one: as soon as it encounters the first, it jumps over the rest of the try block and starts to check the except blocks.

# Itertools



This tool create all possible 2 elt combination the list.

## Difference between extend and append