1. Glossary

REPL – Read-Evaluate-Print-Loop (python’s shell)

1. Introduction

 Python is an **interpreted language**. That means that, unlike languages like C and its variants, Python does not need to be compiled before it is run. Other interpreted languages include PHP and Ruby.

 Python is **dynamically typed**, this means that you don't need to state the types of variables when you declare them or anything like that. You can do things like x=111 and then x="I'm a string" without error

 Python is well suited to object orientated programming in that it allows the definition of classes along with composition and inheritance. Python does not have access specifiers (like C++'s public, private), the justification for this point is given as "we are all adults here"

 In Python, functions are first-class objects. This means that they can be assigned to variables, returned from other functions and passed into functions. Classes are also first class objects

 Writing Python code is quick but running it is often slower than compiled languages. Fortunately， Python allows the inclusion of C based extensions so bottlenecks can be optimised away and often are. The numpy package is a good example of this, it's really quite quick because a lot of the number crunching it does isn't actually done by Python

 Python finds use in many spheres - web applications, automation, scientific modelling, big data applications and many more. It's also often used as "glue" code to get other languages and components to play nice.

 [Python makes difficult things easy](https://xkcd.com/353/) so programmers can focus on overriding algorithms and structures rather than nitty-gritty low level details

1. Definitions

## Duck Typing

"If it looks like a duck and quacks like a duck, it's a duck"

**In other words, we are interested in what the object can do, rather than with what the object is.**

The idea is that you don't look at the type in order to know if an object support a function. You try calling the function anyway. If it’s defined, it will work. If it doesn’t, it will throw an exception.

For example:

def f(x):

x.Quack()

If f gets an x supporting a Quack(), everything is fine, if not, it will crash at runtime.

But duck typing doesn't imply dynamic typing at all - in fact, there is a very popular but completely static duck typing approach that doesn't give any requirements too:

template <typename T>

void f(T x) { x.Quack(); }

The function doesn't tell in any way that it wants some x that can Quack, so instead it just tries *at compile time* and if everything works, it's fine.

## CPython

CPython is the original Python implementation. It is the implementation you download from Python.org. People call it CPython to distinguish it from other, later, Python implementations, and to distinguish the implementation of the language engine from the Python programming language itself.

The latter part is where your confusion comes from; you need to keep Python-the-language separate from whatever runs the Python code.

CPython happens to be implemented in C. That is just an implementation detail, really. CPython compiles your Python code into bytecode (transparently) and interprets that bytecode in a evaluation loop.

CPython is also the first to implement new features; Python-the-language development uses CPython as the base; other implementations follow.

### What about Jython, etc.?

[Jython](http://www.jython.org/), [IronPython](http://ironpython.net/) and [PyPy](https://pypy.org/) are the current "other" implementations of the Python programming language; these are implemented in Java, C# and RPython (a subset of Python), respectively. Jython compiles your Python code to Java bytecode, so your Python code can run on the JVM. IronPython lets you run Python on the [Microsoft CLR](https://docs.microsoft.com/en-us/dotnet/standard/clr). And PyPy, being implemented in (a subset of) Python, lets you run Python code faster than CPython, which rightly should blow your mind. :-)

### Actually compiling to C

So CPython does **not** translate your Python code to C by itself. It instead runs a interpreter loop. There is a project that does translate Python-ish code to C, and that is called [Cython](http://cython.org/). Cython adds a few extensions to the Python language, and lets you compile your code to C extensions, code that plugs into the CPython interpreter.

# Style Guide

import this

<https://www.python.org/dev/peps/pep-0008/>

## Docstring

(from PEP8): Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the def line.

<https://google.github.io/styleguide/pyguide.html>

* At the start of every module, before any imports:

"""Retrieve and print words from a URL

Usage:

python3 words.py <URL>

"""

* At the start of every public function before any code:

"""Fetch a list of words from a URL

Args:

url: the URL of a UTF-8 text document

Returns:

A list of strings containint the words from the document.

"""

## Naming convention

* ClassName: Camal Case
* \_single\_leading\_underscore: weak "internal use" indicator. E.g. from M import \* does not import objects whose name starts with an underscore.  
  Also used by convention to indicate a private class member or method that shouldn’t be accessed from the outside.
* single\_trailing\_underscore\_: used by convention to avoid conflicts with Python keyword, e.g.
* Tkinter.Toplevel(master, class\_='ClassName')
* \_\_double\_leading\_underscore: when naming a class attribute, invokes name mangling (inside class FooBar, \_\_boo becomes \_FooBar\_\_boo; see below).
* \_\_double\_leading\_and\_trailing\_underscore\_\_: "magic" objects or attributes that live in user-controlled namespaces. E.g. \_\_init\_\_, \_\_import\_\_ or \_\_file\_\_. Never invent such names; only use them as documented.
* \_ - should be used in comprehensions or assignments (e.g. tuple unpacking) to discard the item we don’t use.

## Modules

Related classes and global functions are usually grouped together in the same module.

# PyPI – the Python Package Index

**To install a package** from the index:

* Either [pip](https://pip.pypa.io/) install package ([get pip](https://pip.pypa.io/en/latest/installing.html)) or
* download, unpack and "python setup.py install" it.

To search a package:

Pip search <package name search expression>

# Idle

* To start python 3 on windows: >>py
* To start python 3 on Linux: >> python3
* You can pass -m to your python commands to have it run a module as a script (e.g. python -m venv or python -m pdb).
* To re-load a module (for debugging):

>>> from importlib import reload

>>> reload(<module>)

* To find out where a module’s source file is located:

import <module>  
<module>.\_\_file\_\_

## Help

* To open the interactive help console (from idle):

help()

* To get help on specific object or module:

help(object)

Note: if the module is not part of the default library, you will have to import the module before calling help on the module.

# Data Types

Everything in python is an object and access to it is through a pointer!

* To pretty-print all built in data types, we can use the function pprint from the standard module pprint (pretty printing):  
  # The following import as is important otherwise we  
  # won’t be able to access the function that has the   
  # same name as the module!!  
  from pprint import pprint as pp   
  pp(d1)

## Mutable vs Immutable Types

* Immutable types:
  + numbers,
  + strings and
  + tuples
* Mutable Types:
  + dictionaries and
  + lists: queue, stack
  + classes. You can change a class’s methods in runtime (see Monkey Typing), although other than in unit testing, very bad practice.

## str

* python 3 automatically concatenates strings into one:  
  >>> “first string” “ and 2nd string” => “first string and 2nd string”
* r’this is a raw string’
* str() – convert other types to strings.
* Bytes: a special type of string that encode the data as bytes instead of utf-8 like the string’s default:  
  b’This is a bytes string’
  + To translate from str to bytes: use ‘encode’
  + To translate from bytes to string: use ‘decode’
  + Files, Network resources and HTTP responses are transmitted as bytes streams and therefore, they need to be decoded before working on them as strings.
* **Join**: to concatenate many strings, using join is more efficient than using the ‘+’. Call the **join()** method on the **separator** string:  
  >>> colors = ‘;’.join([‘red’, ‘blue’, ‘green’])  
  ‘red;blue;green’
* To split strings, use the **split()** method with the separator again:  
  >>> colors.split(';')  
  ['red', 'blue', 'green']
* A quick way to concatenate strings is to join() on a empty string:  
  >>> ''.join(['high', 'way', 'man'])  
  'highwayman'
* **partition()** method divides a string into three (returned as a tuple) around a separator: prefix, separator and suffix, returning a tuple:

>>> departure, seperator, arrival = "London:Edinburgh".partition(':')

If we don’t want to use the separator, the convention is to assign it to ‘\_’:  
>>> origin, \_, destination = "Seattle-Boston".partition('-')

* **format()**
  + can use pre-defined tuples:

>>> pos = (32.5, 89, 13.7)

>>> "Three Numbers: {pos[0]} , {pos[1]} and {pos[2]}".format(pos=pos)

'Three Numbers: 32.5 , 89 and 13.7'

* + can even use object fields:

>>> import math

>>> "Pi = {m.pi:.3f}, e={m.e:.3f}".format(m=math)

'Pi = 3.142, e=2.718'

## Range

Rage is actually a collection type rather then a container.

* Don’t use range() to iterate over lists! This is un-pythonic! Use direct iteration instead:
  + NO:

>>> s = [0, 1, 2, 3]

>>> for i in range(len(s)):

... print(s[i])

* + Yes:

>>> for val in s:

... print(val)

* Prefer enumerate() for counters:  
  enumerate() yields (index, value) tuples:

>>> t = [6, 735, 9045, 59]

>>> for p in enumerate(t):

... print(p)

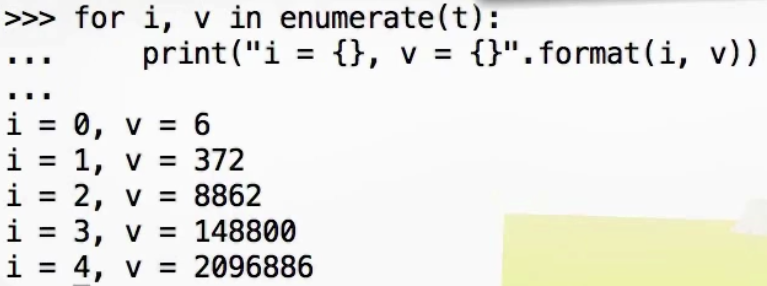
...

(0, 6)

(1, 735)

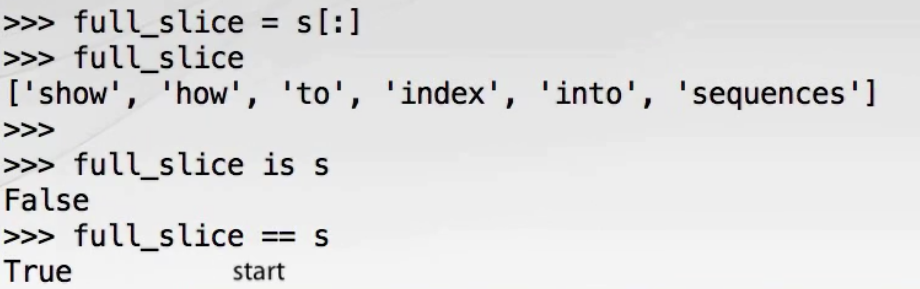
(2, 9045)

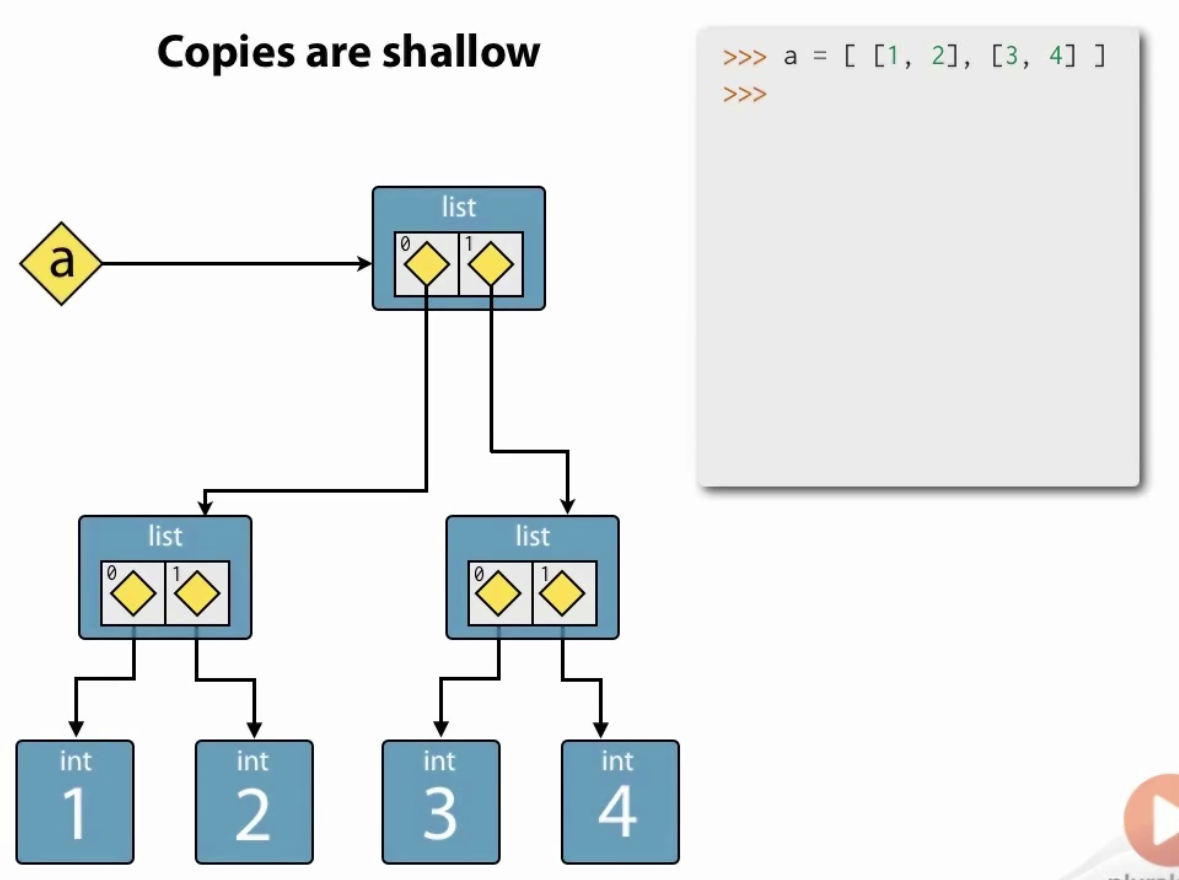
(3, 59)

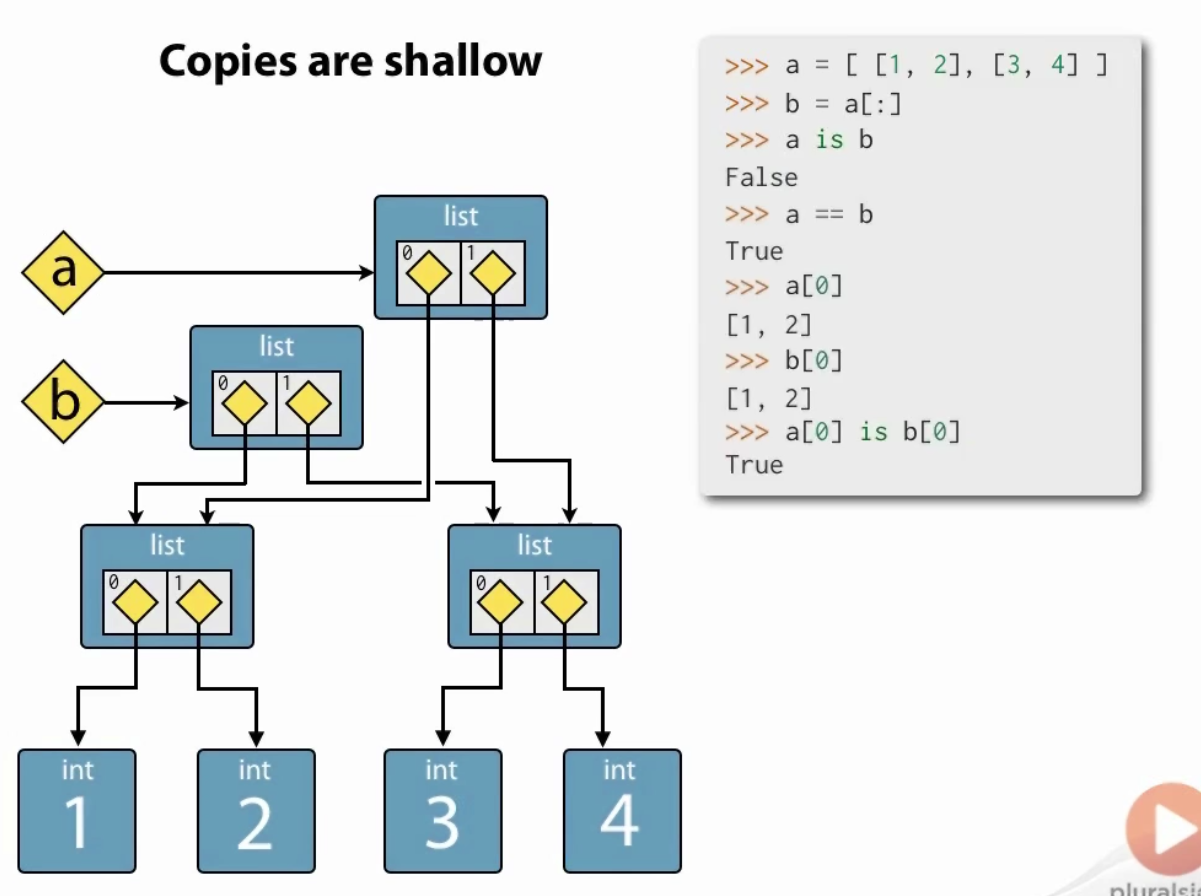
* Or with tuple unpacking:  
  

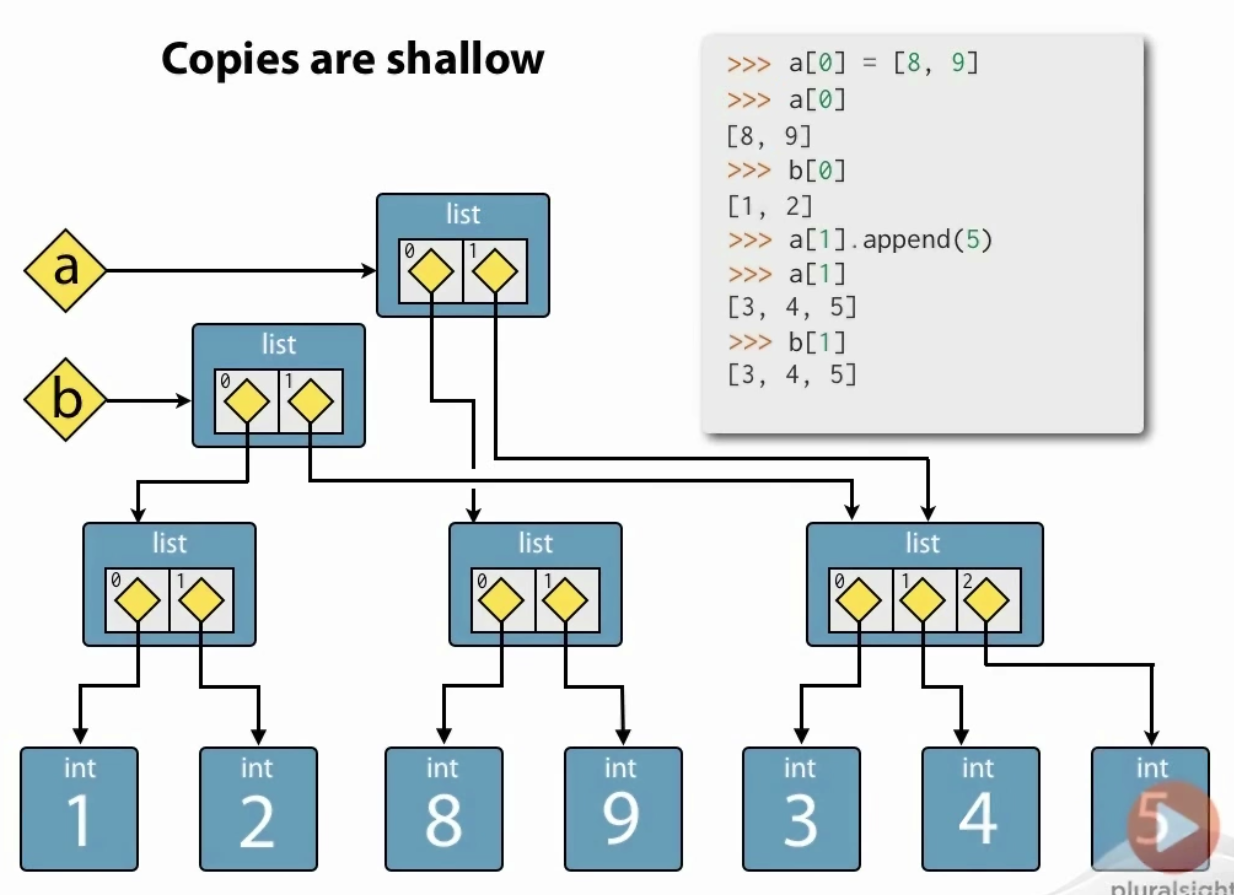
## Lists

List = []

* To index from the end of the list, use negative indexes (useful also for tuples). The last item in the array is index **-1**
* Full slicing will copy the list:
* More readable ways to copy a list are with:
  + The copy method: t = s.copy()
  + List constructor: t = list(s)  
    The advantage of using the copy constructor is that it will work with any iterable type and not just lists.
* NOTE: all the copies above are shallow! They copy the references to the objects pointed to by the list, they don’t copy the objects themselves!!







* Repetition using the operator ‘\*’:

>>> lst = [1,2] \* 5   
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]

**Note: like copy, list repetition is shallow!!**

* lst.index(item) - will return the integer index of the first equivalent element and raises ‘ValueError’ if not found:
* lst .cout(item) – how many of the given item.
* In / not in – test for memebership
* del lst [indx] - remove the item in index <index>
* lst.remove(val) – remove item with value <val> from the list. Like writing: del lst(lst.indx(val))
* insert(indx,val)
* concatenate lists with ‘+’
* Extend a list in place with ‘+=’ or extend() method.
* lst.reverse() – reverse the list in place
* sort() – sort the list in place. we can pass any callable object as a key that can will be called on every item in the list and the returned value will be used to sort the list. For example:

>> lst = ['This', 'is', 'a', 'list', 'of', 'words']

>> lst.sort()

>> print(lst)  
['a', 'is', 'of', 'This', 'list', 'words']

* sorted(lst) : sort any iterable series (not in-place) and returns a list
* reversed(lst) : reverses any iterable series and returns a reverse iterator

### List Comprehension

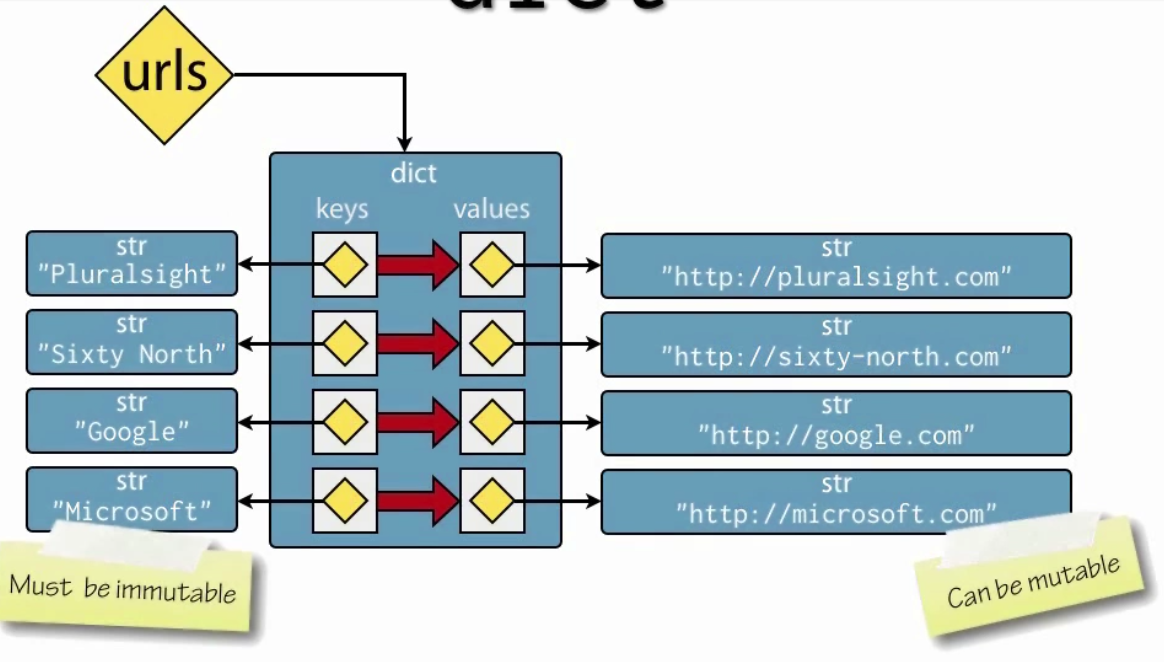
**[exp(item) for item in iterable if predicate(item)]**

List = [“Senior" if age >= 55 and handicap >= 8 else "Open" for (age, handicap) in data]

def tower\_builder(n\_floors):  
 max\_stars = (n\_floors\*2) – 1  
 return [''.join('\*'\*times).center(max\_stars,  
 ' ') for times in range(1,max\_stars+1,2)]

List = [<an expression that result in *the member to   
 add into the array*>   
 **for**  
 <an expression that results in *an iterator*>]

## Dictionaries



* Dictionaries are mutable
* Keys are immutable
* Values are immutable

In order to avoid python creating new keys when the user have typos, it is better to define the keys outside the dictionary and use the defines in the dictionary instead of using direct strings:

class StudentKeys:

NAME = "Name"

SURNAME = "Surname"

…

Students = [{StudentKeys.NAME : “John”,

StudentKeys.SURNAME : “Smith”},

{StudentKeys.NAME : “June”,

StudentKeys.SURNAME : “Fry”}]

* To iterate over the keys of the dictionary:

Colors = {…}  
for color in colors: is the same as writing   
for key in colors.keys():

* To iterate over the values of the dictionary:

for value in d.values():

* To iterate over the key-value tuples:

For key, value in d.items():

* In/not in operators work only on the keys!
* dict() : construct a dictionary from:
  + a list of key-value tuples:   
    d1 = dict([(‘key1’, 5), (‘key2’,6)])
  + key-value pairs:  
    d2 = dict(key1=7, key2=8)
* copying dictionaries is shallow like all other object types!
  + d.copy()
  + e = dict(d)
* To extend a dictionary with all the values form another dictionary:
  + d2.update(d1)

If there are duplicated keys in the original and the added dictionary, the added dictionary will overwrite the original dictionary’s values.

* Delete a value:

del d[‘key1’]

### Dictionary Comprehension

**{key\_exp(item):value\_exp(item) for item in iterable if predicate(item)}**

Note: later keys overwrite earlier keys!

## Tuple

Heterogeneous immutable sequence.

t = (1,”name”,5.7)

to create a single element, add a comma at the end:

t = (1,)

empty tuple:

t = ()

concatenate with +; multiply with \*

Tuples are useful for multiple return values:

def minmax(items):  
 return min(items), max(items)

the interpreter packs this automatically into a tuple. No need to add the parenthesis.

Tuple unpacking allows us to de-structure directly into named references:

lower, upper = minmax([23, 56, 72, 1])

Tuple unpacking also works with arbitrarily nested tuples (although not with other data structures):  
>>> (a, (b, (c, d))) = (4, (3, (2, 1)))

This allows for the simple python swapping of elements:

>>> a, b = b, a

Tuple constructor:

>>> tuple([1, 2, 3, 4]) => (1, 2, 3, 4)

>>> tuple(“string”) => ('s', 't', 'r', 'i', 'n', 'g')

Membership testing: use in and not in operators

## Set

Unordered set of immutable objects.

* To create an empty set: set()
* set() constructor accepts:
  + Iterable series of values
  + Duplicates are discarded
  + Often used specifically in order to remove duplicates.   
    Note: this will not preserve the order!
* Note: iterating over a set will be in arbitrary order!
* To remove an element from the set:
  + s1.remove(val) : will through a ‘KeyError’ exception if the value is not found.
  + s1.discard(val) : always succeeds.
* Set algebra:
  + Union: s.union(t) . all the elements in one of the sets or in both. Commutative (we can change the order).
  + Intersection: s.intersection(t). all the elements that exist in both sets. Commutative
  + Difference: s.difference(t). all the elements that are in the 1st set but are not in the 2nd set. Non-commutative
  + Symmetric difference: s.symmetric\_difference(t). all the elements that are in one set or in the other set but not in both. Commutative.
  + s.issubset(t) : is s is a subset of set t?
  + s.issuperset(t): is s a super set of set t?
  + s.disjoint(t): does set s and set t don’t have any elements in common?

### Set Comprehension

**{exp(item) for item in iterable if predicate(item)}**

Note: duplicates will be eliminated since this is a set. In addition, the order of the items inside the set is not guaranteed.

### FrozenSet

The same as set, only immutable. This means that it can be used as keys for dictionaries (which set can’t be).

## Generators

* Specify iterable sequences
  + All generators are iterators
* Are lazily evaluated
  + The next value in the sequence is only computed on demand
* (and therefore) Can model infinite sequences
  + Such as data streams with no definite end, mathematical series, sensor readings etc.
* Are composable into pipelines
  + For natural stream processing
* Generator is any python function that has the yield keyword at least once in its body. It can also have the return statement without any parameter (or the implicit return statement at the end):

def gen123():  
 yield 1  
 yield 2  
 yield 3  
 return

* The generator will return the next value with the yield statement and will resume its execution in the line immediately after this line when it’s called next.
* Each call to a generator function will create a new generator object. We can use this generator to create and iterate over all the values.

### Generator Comprehension

Create a generator object using comprehension and lazy evaluation:

**(exp(item) for item in iterable if predicate(item))**

Note: generators are single-use object. Once they’ve finished yielding all their values, they will be empty:

million\_squares = ( x\*x for x in range(1,100001))

print("At this point none of the square has been created. Million\_squares is a generator: {}".format(million\_squares))

print("Force generation of the full range: {}".format(list(million\_squares)))  
=> [1,4,9,….,100000000000]

print("Now the generator is empty: {}".format(list(million\_squares)))  
=> []

We can also create the generator in a function call:

sum(x\*x for x in range(1,1000001) if is\_prime(x))

## Files

* File objects are iterators and can be used to iterate over the file line by line.
* Make sure to always close opened file to prevent resource leak.
* Files are context managers and therefore, can use the with-statement to ensure that clean up operations, such as closing the file, are performed even if there was an exception.
* The notion of file-like objects is loosely defined, but very useful in practice.
  + Exercise EAFP (Easier to Ask Forgiveness than Permission) to make the most of them.
* File-like objects:
  + http

## \* and \*\*

<https://treyhunner.com/2018/10/asterisks-in-python-what-they-are-and-how-to-use-them/>

def func(\*args):

\*args – accept any number of arguments separately or in a list

def func(\*\*kwargs):

\*\*kwargs – accept a dictionary of keys and values or a list of keys and values.

## Lambda

lambda arguments : expression

The expression is executes, and the result is returned

Example:  
x = lambda a : a + 10  
print(x(5))

# Protocol

In Python in order to support a specific protocol, the object type just need to implement the required functions. There’s no need to inherit from certain objects etc.

|  |  |  |
| --- | --- | --- |
| Protocol Name | Required to Implement | Collections that implement this protocol |
| Container | Checking if an element is in the collection with in and not in | str, list, range, tuple, bytes, set, dict |
| Sized | len(obj) | str, list, range, tuple, bytes, set, dict |
| iterable | Can produce an iterator with iter(s):  For item in iterable:  do\_somthing(item) | str, list, range, tuple, bytes, set, dict |
| Sequence | * Retrieve elements by index: item = seq[index] * Find items by value: index = seq.index(item) * Count items: num = seq.count(item) * Produce a reversed sequence: r = reversed(seq) | str, list, range, tuple, bytes, set, dict |
| Mutable Sequence |  | list |
| Mutable Set |  | set |
| Mutable Mapping |  | dict |

## Iteration Protocols

* See itertools module for iteration functions and tools.
* Iterable Protocol:

Iterable objects can be passed to the built-in **iter()** function to get an iterator:

Iterator = iter(iterable)

This will return the iterator itself. To get the 1st item in the collection, call next(iterator) – see below.

* Iterator Protocol:

Iterator objects can be passed to the built-in **next()** function to get the next item:

Item = next(iterator)  
  
Calling next(iterator) on the last element in the iterator, will return a StopIteration exception.

# Built in Functions

// - divide as integers: 5//3 will return 1 instead of 1.66

**bool**([]) =>False ; bool([3,5]) =>False

bool(“”) =>False ; bool(“False”) =>True

**zip()** - Returns an iterator of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted.

x = [1, 2, 3]

>>> y = [4, 5, 6]

>>> zipped = zip(x, y) # zipped = (1,4) (2,5)…

>>> list(zipped)

[(1, 4), (2, 5), (3, 6)]

Map() –

**any(iterable)**: is any of the iterable objects evaluate to True?

any([False, True, False])  
=> True  
any(is\_prime(x) for  
=> False

**all(iterable)**: does all the iterable objects evaluate to True?

all([False, True, False])  
=> Falsee  
all(name == name.title() for name in ['London', 'New York', 'Sydney'])  
=> True

**enumerate(iterable, start=0)**:  
Return an enumerate object of tuples (count, value) where count start from ‘start’ and value is the value from the iterable object:

>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']  
>>> list(enumerate(seasons))  
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]

**filter(function, iterable):**

Construct an iterator from those elements of iterable for which function returns true.

filter(function , iterable) is equivalent to:  
(item for item in iterable if function(item))

**map(function, iterable,…):**  
Return an iterator that applies function to every item of iterable, yielding the results.

## Default Arguments

Python’s default arguments are evaluated once when the function def is evaluated (on module import), not each time the function is called (like it is in say, Ruby). This means that if you use a mutable default argument and mutate it, you will and have mutated that object for all future calls to the function as well.

If you need to use mutable values as default parameters, a way to work around this issue is to use ‘None’ instead. For example:

Instead of writing:

def append\_to\_list(value, list = []):

list.append(value)

Use:

def append\_to\_list(value, list = None):

if not list:

list = []

list.append(value)

# Control

## Command Line Arguments

* Python standard library: argparse
* Many third-party options such as: docopt

## With Statement – Context Manager

Whenever we use external resources (e.g. files, sockets etc), we should use the with statement in order to avoid resource leak:

with open(‘file’) as data:  
 for line in data:  
 str\_line = line.decode(‘utf-8’)  
 print(str\_line)

### Use Context Management to Manage Your Class

You can extend your classes to support context management.

Instead of:

r = RegrigaratorRaider()

r.open()

t.take(food)

r.close()

You can write:

From contectlib import closing

with closing(RegrigaratorRaider()) as r:

r.open()

t.take(food)

# Exceptions

* A program should fail loudly and not silently!
* Exceptions can not be ignored! But errors can…
* Exceptions are parts of the python API!  
  They should be documented properly!
* Python uses exceptions pervasively. Many built-in language features depend on them.
* Callers need to know what exceptions to expect and when.
* Use exceptions that users will anticipate. Use built-in types when possible.
* Standard exceptions are often the best choice.
* **Do not guard against type error:** Testing for type or catching type exceptions goes against Python
* Programmer errors should not normally be handled
* Exception objects can be converted to strings for printing using the str() constructor.

## It’s Easier to Ask for Forgiveness than for Permission

* There are two philosophies: (LBYL) Look Before You Leap vs (EAFP) It’s Easier to Ask Forgiveness than Permission.
* Python is strongly in favor of EAFP since it’s put primary logic into the happy path in its most readable form without mixing in the error handling into it. Instead, it does the error handling in a separate path.
* Returning Error Codes from functions require interspersed, local handling.
* Exceptions allow centralized, non-local handling.
* In addition, exceptions require explicit handling otherwise, they terminate the program. This is in contrast to returning error codes which are silent be default and can be easily ignored.

## Raise and Catch

Raise:

raise ValueError(“Invalid input: {}”.format(x))

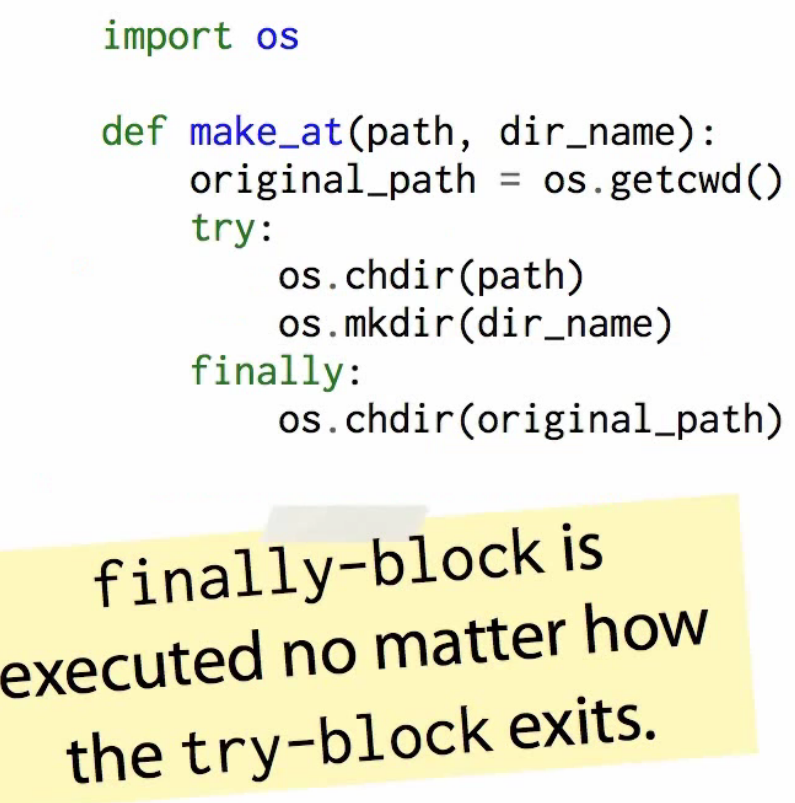
raise without an argument – re-raises the current exception.

try-catch:

try:  
 … code to try  
Except ValueError as e:  
 print(e, file= sys.stderr)  
print(“Program execution continues from here”)

## Resource Cleanup

try…finally lets you clean up whether an exception occurs or not. It can be used in conjunction with except block or on its own:



# Classes

class <ClassName>:

def \_\_init\_\_(self,…):

# NOTE: unlike C# or Java, the object exist fully at this point.

# \_\_init\_\_ is just the initializer of the object and not its constructor!

<do any initiatlizations of the class>

## Initializers

* Python class initializers only initializes an already existing object. They are not constructors in the C++ sense of the word.
* Initializers should establish the invariants of the class and through an exception for every initialization parameter that breech these invariants.
* def \_\_init\_\_(self, val):
* if val <= 0:
* raise ValueError("Invalid initialization value: '{}'".format(val))
* self.\_positive\_value = val

## Inheritance

* Since python has duck-typing, we don’t really need to use inheritance in order to facilitate polymorphism.
* Instead, we use inheritance in order to re-use implementation across different classes.

All common functions can be defined in the parent with only the different implementations defined in the children classes.

* All methods from the base class are inherited by its sub-classes including all special methods like the \_\_init\_\_.

## Multiple Inheritance

Method Resolution Order, or MRO

for new-style classes in Python 2.2, I adopted a scheme where the MRO would be pre-computed when a class was defined and stored as an attribute of each class object. The computation of the MRO was officially documented as using a depth-first left-to-right traversal of the classes as before. If any class was duplicated in this search, all but the last occurrence would be deleted from the MRO list.

* + - 1. Calculate the depth first, left to right MRO.
      2. All classes that are duplicated in this search, all but the last occurrence will be deleted.

class A:

def m(self):

print("m of A called")

class B(A):

def m(self):

print("m of B called")

class C(A):

def m(self):

print("m of C called")

class D(B,C): - **order of the base-classes define which implementation will be called! In this case, B will be called.**

Pass

## Polymorphism

class Animal:

def \_\_init\_\_(self, name): # Constructor of the class  
 self.name = name

# Abstract method, defined by convention only  
 def talk(self):   
 raise NotImplementedError("Subclass must implement abstract method")

class Cat(Animal):

def talk(self):  
 return 'Meow!'

class Dog(Animal):

def talk(self):  
 return 'Woof! Woof!'

animals = [Cat('Missy'),  
 Cat('Mr. Mistoffelees'),  
 Dog('Lassie')]

for animal in animals:  
 print animal.name + ': ' + animal.talk()

# prints the following:

#

# Missy: Meow!

# Mr. Mistoffelees: Meow!

# Lassie: Woof! Woof!

# Garbage Collector

* Python maintains a count of the number of references to each object in memory. If a reference count goes to zero then the associated object is no longer live and the memory allocated to that object can be freed up for something else
* occasionally things called "reference cycles" happen. The garbage collector periodically looks for these and cleans them up. An example would be if you have two objects o1 and o2 such that o1.x == o2 and o2.x == o1. If o1 and o2 are not referenced by anything else then they shouldn't be live. But each of them has a reference count of 1.
* Certain heuristics are used to speed up garbage collection. For example, recently created objects are more likely to be dead. As objects are created, the garbage collector assigns them to generations. Each object gets one generation, and younger generations are dealt with first.

# Decorators

A decorator is a special kind of function that either takes a function and returns a function, or takes a class and returns a class. The @ symbol is just syntactic sugar that allows you to decorate something in a way that's easy to read.

@my\_decorator

def my\_func(stuff):

do\_things

## Closures

In python, an inner function that is defined within another function remember its enclosing scopes **at definition time.** This means that if we invoke that without the scope of the outer function, they will still have them and will be able to user them as part of their scope resolution process.

>>> **def** **outer**(x):

... **def** **inner**():

... **print** x *# 1*

... **return** inner

>>> print1 = **outer**(1)

>>> print2 = **outer**(2)

>>> **print1**()

1

>>> **print2**()

2

We see here that print1 and print2 keep a snapshot of outer at the time when they were defined. Print 1 remember outer with x=1 and print2 remembers it with x=2.

## Decorators

We can use Python’s closures to decorate functions:

>>> **def** **outer**(some\_func):

... **def** **inner**():

... **print** "before some\_func"

... ret = **some\_func**() *# 1*

... **return** ret + 1

... **return** inner

>>> **def** **foo**():

... **return** 1

>>> decorated = **outer**(foo) *# 2*

>>> **decorated**()

before some\_func

2

We can also re-assigning the original function to the decorated one:

>>> foo = **outer**(foo)

>>> foo *# doctest: +ELLIPSIS*

<function inner at 0x...>

## @

@ is Python syntactic sugar for assigning a function to it’s decorated version:

Instead of writing:

>>> add = **wrapper**(add)

To decorate add with the wrapper decorator, we can use the @ syntax in the following way:

>>> @wrapper

... **def** **add**(a, b):

... **return** **Coordinate**(a.x + b.x, a.y + b.y)

This pattern can be used at any time, to wrap any function. But if we are defining a function we can "decorate" it with the @ symbol. It’s important to recognize that this is no different than simply replacing the original variable add with the return from the wrapper function - Python just adds some syntactic sugar to make what is going on very explicit.

# Import

* Import <module>
* from my\_module import (function1, function2,….)
* from <module> import \* - caution – never use this in a program!!!  
  This is useful if you want to debug stuff from the REPL, allowing you full access to all your script’s functions.
* To re-import a previously imported module in idle:
  + import importlib, then do importlib.reload(nameOfModule).

#### Basic Definitions

* **module**: any \*.py file. Its name is the file name.
* **built-in module**: a “module” (written in C) that is compiled into the Python interpreter, and therefore does not have a \*.py file.
* **package**: any folder containing a file named \_\_init\_\_.py in it. Its name is the name of the folder.
  + in Python 3.3 and above, any folder (even without a \_\_init\_\_.py file) is considered a package
* **object**: in Python, almost everything is an object - functions, classes, variables, etc.

#### Example Directory Structure

test/ # root folder

packA/ # package packA

subA/ # subpackage subA

\_\_init\_\_.py

sa1.py

sa2.py

\_\_init\_\_.py

a1.py

a2.py

packB/ # package packB (implicit namespace package)

b1.py

b2.py

math.py

random.py

other.py

start.py

According to Python documentation, here is how an import statement searches for the correct module or package to import:

When a module named spam is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named spam.py in a list of directories given by the variable sys.path. sys.path is initialized from these locations:

* The directory containing the input script (or the current directory when no file is specified).
* PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).
* The installation-dependent default.

After initialization, Python programs can modify sys.path. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory.Source: Python [*2*](https://docs.python.org/2/tutorial/modules.html#the-module-search-path) and [*3*](https://docs.python.org/3/tutorial/modules.html#the-module-search-path)

Let’s recap the order in which Python searches for modules to import:

1. modules in the Python Standard Library (e.g. math, os)
2. modules or packages in a directory specified by sys.path:
   1. If the Python interpreter is run interactively:
      * sys.path[0] is the empty string ''. This tells Python to search the current working directory from which you launched the interpreter, i.e. the output of pwd on Unix systems.

If we run a script with python <script>.py:

* + - sys.path[0] is the path to <script>.py
  1. directories in the PYTHONPATH environment variable
  2. default sys.path locations

#### All about \_\_init\_\_.py

An \_\_init\_\_.py file has 2 functions.

1. convert a folder of scripts into an importable package of modules (before Python 3.3)
2. run package initialization code

#### Converting a folder of scripts into an importable package of modules

In order to import a module or package from a directory that is not in the same directory as the script we are writing (or the directory from which we run the Python interactive interpreter), that module needs to be in a package.

As defined above, any directory with a file named \_\_init\_\_.py is a Python package. This file can be empty. For example, when running Python 2.7, start.py can import the package packA but not packB because there is no \_\_init\_\_.py file in the test/packB/ directory.

This does NOT apply to Python 3.3 and above, thanks to the adoption of implicit namespace packages. Basically, Python 3.3+ treats all folders as packages, so empty \_\_init\_\_.py files are no longer necessary and can be omitted.

* Note: if a1.py calls import a2 and we run python a1.py, then test/packA/\_\_init\_\_.py will NOT be called, even though it seems like a2 is part of the packA package. This is because when Python runs a script (in this case a1.py), its containing folder is not considered a package.

#### Using Objects from the Imported Module or Package

There are 4 different syntaxes for writing import statements.

1. import <package>
2. import <module>
3. from <package> import <module or subpackage or object>
4. from <module> import <object>

Let X be whatever name comes after import.

* If X is the name of a module or package, then to use objects defined in X, you have to write X.object.
* If X is a variable name, then it can be used directly.
* If X is a function name, then it can be invoked with X()

Optionally, as Y can be added after any import X statement: import X as Y. This renames X to Y within the script. Note that the name X itself is no longer valid. A common example is import numpy as np.

The argument to the import function can be a single name, or a list of multiple names. Each of these names can be optionally renamed via as. For example, this is would be a valid import statement in start.py: import packA as pA, packA.a1, packA.subA.sa1 as sa1

Example: start.py needs to import the helloWorld() function in sa1.py

* Solution 1: from packA.subA.sa1 import helloWorld
  + we can call the function directly by name: x = helloWorld()
* Solution 2: from packA.subA import sa1 or equivalently import packA.subA.sa1 as sa1
  + we have to prefix the function name with the name of the module: x = sa1.helloWorld()
  + This is sometimes preferred over Solution 1 in order to make it explicit that we are calling the helloWorld function from the sa1 module.
* Solution 3: import packA.subA.sa1.
  + we need to use the full path: x = packA.subA.sa1.helloWorld()

#### Use dir() to examine the contents of an imported module

After importing a module, use the dir() function to get a list of accessible names from the module. For example, suppose I import sa1. If sa1.py defines a helloWorld() function, then dir(sa1) would include helloWorld.

#### Importing Packages

Importing a package is conceptually equivalent to importing the package’s \_\_init\_\_.py file as a module. Indeed, this is what Python treats the package as:

**>>>** **import** packA

**>>>** packA

**<**module 'packA' **from** 'packA\\_\_init\_\_.py'**>**

Only objects declared in the imported package’s \_\_init\_\_.py are accessible to the importer. For example, since packB lacks a \_\_init\_\_.py file, calling import packB (in Python 3.3+) has very little use because no objects in the packB package are made available. A subsequent call to packB.b1 would fail because it has not been imported yet.

#### Absolute vs. Relative Import

An **absolute import** uses the full path (starting from the project’s root folder) to the desired module to import.

A **relative import** uses the relative path (starting from the path of the current module) to the desired desired module to import. There are two types of relative imports:

* an explicit relative import follows the format from .<module/package> import X, where <module/package> is prefixed by dots . that indicate how many directories upwards to traverse. A single dot . corresponds to the current directory; two dots .. indicate one folder up; etc.
* an implicit relative import is written as if the current directoy is part of sys.path. **Implicit relative imports are only supported in Python 2. They are NOT SUPPORTED IN PYTHON 3.**

The Python documentation says the following about Python 3’s handling of relative imports:

The only acceptable syntax for relative imports is from .[module] import name. All import forms not starting with . are interpreted as absolute imports.

Source: [What’s New in Python 3.0](https://docs.python.org/3.0/whatsnew/3.0.html)

For example, suppose we are running start.py which imports a1 which in turn imports other, a2, and sa1. Then the import statements in a1.py would look as follows:

* absolute imports:
* **import** other
* **import** packA.a2
* **import** packA.subA.sa1
* explicit relative imports:
* **import** other
* **from** . **import** a2
* **from** .subA **import** sa1
* implicit relative imports (NOT SUPPORTED IN PYTHON 3):
* **import** other
* **import** a2
* **import** subA.sa1

Note that for relative imports, the dots . can go up only up to (but not including) the directory containing the script run from the command line. Thus, from .. import other is invalid in a1.py. Doing so results in the error ValueError: attempted relative import beyond top-level package.

In general, absolute imports are preferred over relative imports. They avoid the confusion between explicit vs. implicit relative imports. In addition, any script that uses explicit relative imports cannot be run directly:

Note that relative imports are based on the name of the current module. Since the name of the main module is always “**main**”, modules intended for use as the main module of a Python application must always use absolute imports.

Source: Python [*2*](https://docs.python.org/2/tutorial/modules.html#intra-package-references) and [*3*](https://docs.python.org/3/tutorial/modules.html#intra-package-references)

#### How to Import When sys.path can Change:

Often, we want to be flexible in how we use a Python script, whether run directly on the command line or imported as a module into another script. As shown below, this is where we run into problems, especially on Python 3.

**Example**: Suppose start.py needs to import a2 which needs to import sa2. Assume that start.py is always run directly, never imported. We also want to be able to run a2 on its own.

Let’s summarize our findings about the import statement in a2.py:

| **Run** | **from packA.subA import sa2** | **from subA import sa2** |
| --- | --- | --- |
| start.py | OK | Py2 OK, Py3 fail (subA not in test/) |
| a2.py | fail (packA not in test/packA/) | OK |

For completeness sake, I also tried using relative imports: from .subA import sa2. This matches the result of from packA.subA import sa2.

**Solutions (Workarounds)**: I am unaware of a clean solution to this problem. Here are some workarounds:

1. Use absolute imports rooted at the test/ directory (i.e. middle column in the table above). This guarantees that running start.py directly will always work. In order to run a2.py directly, run it as an imported module instead of as a script:
   1. change directories to test/ in the console
   2. python -m packA.a2
2. Use absolute imports rooted at the test/ directory (i.e. middle column in the table above). This guarantees that running start.py directly will always work. In order to run a2.py directly, we can modify sys.path in a2.py to include test/packA/, before sa2 is imported.
3. **import** os, sys
4. sys**.**path**.**append(os**.**path**.**dirname(os**.**path**.**dirname(os**.**path**.**realpath(\_\_file\_\_))))
5. *# now this works, even when a2.py is run directly*
6. **from** packA.subA **import** sa2

NOTE: This method usually works. However, under some Python installations, the \_\_file\_\_ variable might not be correct. In this case, we would need to use the Python built-in inspect package. See [this StackOverflow answer](https://stackoverflow.com/a/11158224) for instructions.

#### Python 2 vs. Python 3

The most important differences between how Python 2 and Python 3 treat import statements have been documented above. They are re-stated again here, along with some other less important differences.

1. Python 2 supports implicit relative imports. Python 3 does not.
2. Python 2 requires \_\_init\_\_.py files inside a folder in order for the folder to be considered a package and made importable. In Python 3.3 and above, thanks to its support of implicit namespace packages, all folders are packages regardless of the presence of a \_\_init\_\_.py file.
3. In Python 2, one could write from <module> import \* within a function. In Python 3, the from <module> import \* syntax is only allowed at the module level, no longer inside functions.

# Multi-Threading

Python doesn't allow multi-threading in the truest sense of the word. It has a [multi-threading package](https://docs.python.org/2/library/threading.html) but if you want to multi-thread to speed your code up, then it's usually not a good idea to use it. Python has a construct called the Global Interpreter Lock (GIL). The GIL makes sure that only one of your 'threads' can execute at any one time. A thread acquires the GIL, does a little work, then passes the GIL onto the next thread. This happens very quickly so to the human eye it may seem like your threads are executing in parallel, but they are really just taking turns using the same CPU core. All this GIL passing adds overhead to execution. This means that if you want to make your code run faster then using the threading package often isn't a good idea.

There are reasons to use Python's threading package. If you want to run some things simultaneously, and efficiency is not a concern, then it's totally fine and convenient. Or if you are running code that needs to wait for something (like some IO) then it could make a lot of sense. But the threading library won't let you use extra CPU cores.

Multi-threading can be outsourced to the operating system (by doing multi-processing), some external application that calls your Python code (eg, Spark or Hadoop), or some code that your Python code calls (eg: you could have your Python code call a C function that does the expensive multi-threaded stuff).

# Machine Learning Libraries

## Sklearn.naive\_bayes.GaussianNB

Gaussian Naïve Bayes algorithm

**import** **numpy** **as** **np**

**>>> training\_**X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])

**>>> training\_**Y = np.array([1, 1, 1, 2, 2, 2])

**>>> from** **sklearn.naive\_bayes** **import** GaussianNB

**>>>** clf = GaussianNB()

**>>>** clf.fit(training\_X, training\_Y)

GaussianNB(priors=None)

**>>>** print(clf.predict([[-0.8, -1]]))

[1]

**>>>** clf.score(test\_X, test\_Y) – return the accuracy score of the algorithm. Important: should be run on a different testing-set!!! (not the training set).

# Web Development

## Flask

@app.route('/')

@app.route('/hello')

def HelloWorld():

return 'Hello World'

If the user will send any of these routes to the browser, the decorated application HelloWorld will be called.

### Styling

Flash will automatically look for CSS,JavaScript and media files if we put them in a folder called **static**.

### Templates Rendering

Flask will look for HTML templates under the ***‘/templates’*** directory.

In the HTML templates we can write normal HTML data with the following built-in Flask support:

* {% logical python code %}

Note: since we can’t use indentation inside of HTML code, we must use special keywords to end if and for loops:  
{% end<keyword> %}

For example, {% endif %} , {% endfor %} etc.

* {{ printed python code }}

In our python application, we’ll need to tell Flask where to get the data for the temple from. This is called Template Rendering:

from Flask import Flask, render\_template

render\_template(‘our\_html\_template.html’,   
<a list of template parameters and their assignment)

### URL Building

url\_for()

### Flash

In order to show messages in the web application, we can use flash.

For this:

* + - 1. Import flash
      2. In the main():

Create an app.secret\_key to sign the session’s cookies.

* + - 1. In the HTML file:

{% with messages = get\_flashed\_messages() %}

{% if messages %}

<ul>

{% for message in messages %}

<li><strong>{{message}}</strong></li>

{% endfor %}

</ul>

{% endif %}

{% endwith %}

### JSON

To supply **Rest API**, we can use jsonify module:

From flask Import jsonity

### Flask Project Structure

For small-size project:

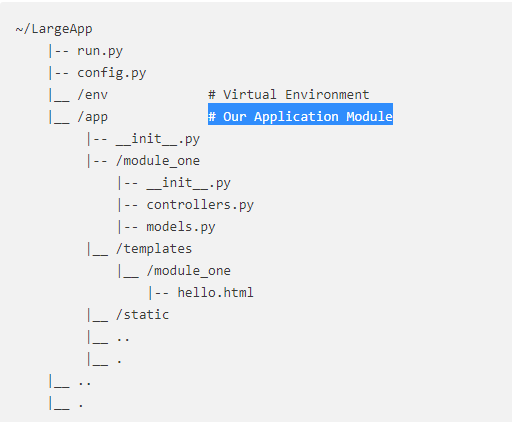
A structure as simple as this would suffice:

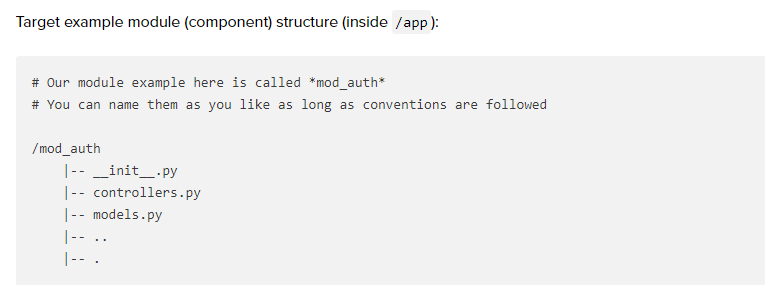
1. Flaskapp/
2. - app.py (controllers)
3. - templates/ (views)
4. - **base**.html (minimum view)
5. - modules.py (**module**/db handling)
6. - **static**/
7. - **base**.css
8. - **base**.js

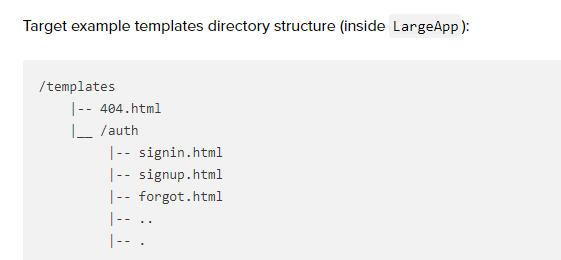
Outside of something like this, you're really stretching what a microframework is great at, which is minimalism. If I needed an mvc-like approach, I'd grab django or pylons. One of the best things about the larger web frameworks is how well tuned they are to building crud apps, and building them quickly.

For major projects - see:

<https://www.digitalocean.com/community/tutorials/how-to-structure-large-flask-applications>



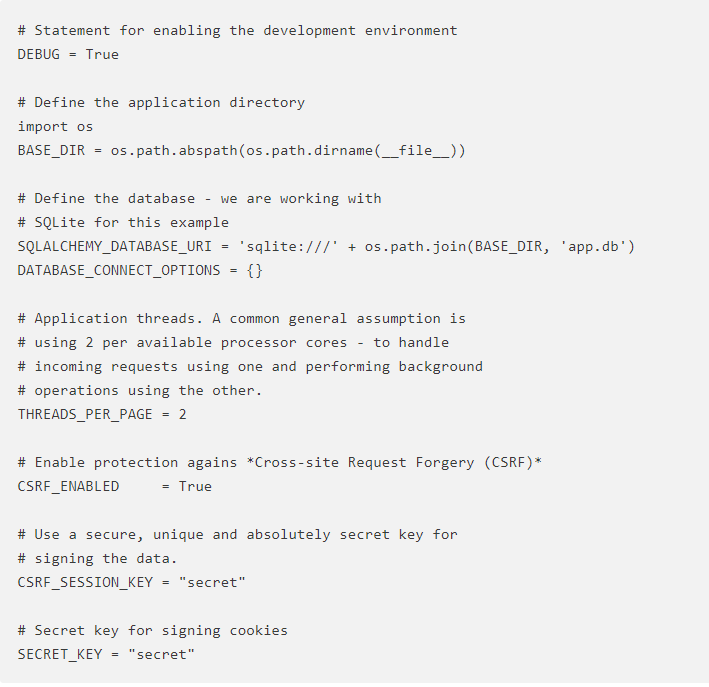




**run.py:**



**config.py:**



### Responder Web Framework

 Flask-like but with async support and

* A pleasant API, with a single import statement.
* Class-based views without inheritance.
* ASGI framework, the future of Python web services.
* WebSocket support!
* The ability to mount any ASGI / WSGI app at a subroute.
* f-string syntax route declaration.
* Mutable response object, passed into each view. No need to return anything.
* Background tasks, spawned off in a ThreadPoolExecutor.
* GraphQL (with GraphiQL) support!
* OpenAPI schema generation.
* Single-page webapp support

 Responder gives you the ability to mount another ASGI / WSGI app at a subroute

 uvicorn: powers responder and is built on top of uvloop

 asgi: <https://www.encode.io/articles/hello-asgi/>

## Jinja2

Jinja2 is a modern day templating language for Python developers. It was made after Django’s template. It is used to create HTML, XML or other markup formats that are returned to the user via an HTTP request.

### Jinja2 syntax in HTML pages:

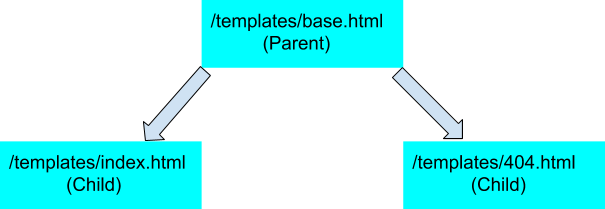
* {%....%} are for executing python statements
  + For loop:  
    {% for name in list\_example %}  
     <li>{{ name }}</li>  
    {% endfor %}
  + To end for loop use {%endfor%} and to end if loop, use {%endif%}
* {{....}} are expressions used to print variable value
* {#....#} are for comments which are not included in the template output
* #....## are used as line statements

### Template inheritance

Jinja 2 supports Template Inheritance, which is one of the most powerful and useful features of any template engine. It means one template can inherit from another template.

Nowadays, websites require the same layout for different pages. Almost every website has a navigation bar attached to its page. To not repeat the code, we use inheritance feature because it saves us a lot of time and also reduces work.

A base template contains the basic layout which is common to all the other templates, and it is from this base template we extend or derive the layout for other pages.



**/app/app.py**

from flask import Flask, render\_template  
   
app = Flask(\_\_name\_\_)  
   
[@app](http://twitter.com/app).route(‘/’)  
def index():  
 return render\_template(‘index.html’)

[@app](http://twitter.com/app).errorhandler(404)  
def page\_not\_found(e):  
 return render\_template(‘404.html’), 404

if \_\_name\_\_ == ‘\_\_main\_\_’:  
 app.run(debug=True)

**/app/templates/base.html**

<! — “Parent Template” →  
<!DOCTYPE html>  
<html>  
<head>  
 **{% block head %}**  
 <title>**{% block title %}{% endblock %}**</title>  
 **{% endblock %}**  
</head>  
<body>  
 {% block body %}{% endblock %}  
</body>  
</html>

Template Inheritance uses {% block %} tag to tell the template engine to **override** the common elements of your site via child templates. base.html is the parent template which is the basic layout and on which you can modify using the child templates, which is used to fill empty blocks with content. base.html is a general layout of the Web page.

**/app/templates/index.html**

<! — “Child Template” →  
**{% extends “base.html” %}**  
{% block title %} Index {% endblock %}  
{% block head %}  
 {{ super() }}  
{% endblock %}  
{% block body %}  
 <h1>Hello World</h1>  
 <p>Welcome to my site.</p>  
{% endblock %}

The {% extend %} must be the first tag in the child templates. This tag tells the template engine that this template extends from the parent template or ( base.html ). {% extend %} represents the inheritance characteristic of Jinja 2. So now the {% extend "base.html"%} first searches for the template mentioned and then the child template index.html overrides it with a different data.

The {% extends %} tag is the key here. It tells the template engine that this template “extends” another template. When the template system evaluates this template, **it first locates the parent**. **The extends tag should be the first tag in the template. Everything before it is printed out normally and may cause confusion.** For details about this behavior and how to take advantage of it, see [Null-Master Fallback](http://jinja.pocoo.org/docs/2.10/tricks/#null-master-fallback).

**/app/templates/404.html**

{% extends ‘base.html’ %}

{% block title %} Page Not Found {% endblock %}

{% block body %}  
 <h1>404 Error :(</h1>  
 <p>What you were looking for is just not there.<p>  
 <a href=”{{ url\_for(‘index’) }}”>go somewhere nice</a>  
{% endblock %}

I’ve also added another child template to the parent template, to show that the layout doesn’t change. This is an error page. So, whenever the user gives an invalid path or web address which does not exist then, 404 Error will pop up. I’ve used an errorhandler() function, which is called when an error happens.

# Virtual Environments

Light-weight, self-contained Python installation.

Included from python 3.3 and up in the standard library with the module venv

To verify that you have venv:

$python3 –m venv

1. To start a new venv environment:  
   $python3 –m venv <new\_directory\_name>

Python will create the new directory and will populate it with the installation.

1. To activate the environment:

* On Linux:

$source <venv\_directory\_name>/bin/activate

* On windows:

> <venv\_directory\_name>/bin/activate

Now, the prompt will change to the environment’s prompt and everything you’ll run will actually run from this environment.

1. To exit the environment:  
   $ deactivate

# Unit Testing

## Unittest

Unittest is the python’s unit test framework. It comes as part of the language package.

The minimal required to work with it:

import unittest

class TestMyClass(unittest.TestCase):

def test\_scenario\_result(self):

obj = some\_object(1,2)

self.assertEqual(2, obj.some\_function())

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

### Test Files

First one should try to **keep the unit tests away from the filesystem** - see this [Set of Unit Testing Rules](http://www.artima.com/weblogs/viewpost.jsp?thread=126923). If possible have your code working with Streams that will be buffers (i.e. in memory) for the unit tests, and FileStream in the production code.

If this is not feasible, you can have your unit tests **generates the files** they need. This makes the test easy to read as everything is in one file. This may also prevent permissions problem.

You can [**mock**](http://en.wikipedia.org/wiki/Mock_object) the filesystem/database/network access in your unit tests.

You can consider the unit tests that rely on DB or file systems as integration tests.

##File:lamb2.py

import sys

def write\_lamb(outfileh):

outfileh.write("Mary had a little lamb.\n")

if \_\_name\_\_ == '\_\_main\_\_':

outfile = open(sys.argv[1])

write\_lamb(outfile)

outfile.close()

##File test\_lamb2.py

import unittest

#import tempfile

import cStringIO

import lamb2

class LambTests(unittest.TestCase):

def test\_lamb\_output(self):

tempfile = cStringIO.StringIO()

# NOTE: Alternatively, for Python 2.6+, you can use

# tempfile.SpooledTemporaryFile, e.g.,

#tempfile = tempfile.SpooledTemporaryFile(10 \*\* 9)

lamb.write\_lamb(tempfile)

expected = "Mary had a little lamb.\n"

tempfile.seek(0)

result = tempfile.read()

self.assertEqual(result, expected)

### Running

#### From Command Line

To run specific tests (separate by spaces)/a whole test class or a whole module, run from command line:

$ python -m unittest test\_module.TestClass.test\_method

#### In Eclipse

Ctrl+F9

## Mocking

### Installing

On python 2.7, install using:

Pip install Mock

On python 3, it’s part of the language standard library.

### Using

<https://docs.python.org/3/library/unittest.mock-examples.html>

example:

**>>> class** **ProductionClass**:

**...**  **def** method(self):

**...**  self.something(1, 2, 3)

**...**  **def** something(self, a, b, c):

**...**  **pass**

**...**

**>>>** real = ProductionClass()

**>>>** real.something = MagicMock(name=’mock\_something’)

**>>>** real.method()

**>>>** real.something.assert\_called\_once\_with(1, 2, 3)

#### Setting Return Values and Attributes

**>>>** mock = Mock()

**>>>** mock.method.return\_value = 3

**>>>** mock.method()

3

The return value can also be set in the constructor:

>>>

**>>>** mock = Mock(return\_value=3)

**>>>** mock()

3

*@mock.patch*(*'Ports.Dongle1\_USB.Dongle1\_USB'*)

def **test\_verify\_response\_seccessful**(*self*,mock\_l2g):

*self*.InitTest(\_\_file\_\_, devices)

mock\_l2g.receive.return\_value = "$*LSA,,PONG,CPU\r\n"*

*self*.testRunner.Open(*'L2G\_Dongle1'*, mock\_l2g)

*self*.testRunner.SendAndGet(*'L2G\_Dongle1'*, *"$LSC,,PING"*)

*self*.testRunner.VerifyResponse( *"$LSA,.\*?,PONG.\*"*)

*self*.GetAndVerifyResults( True)

#### Mocking standard input

import unittest

from unittest.mock import patch

from io import StringIO

def compare\_multi\_line():

    a, b, c = input().strip().split(' ')

    d, e, f = input().strip().split(' ')

    return '%s %s' % (a, d)

def compare\_one\_line():

    a, b, c = input().strip().split(' ')

    return '%s %s' % (a, c)

fake\_input = iter(['1 2 3', '4 5 6']).\_\_next\_\_

class TestCompareSysStdin(unittest.TestCase):

    @patch("builtins.input", fake\_input)

    def test\_compare\_multi\_line(self):

*self*.assertEqual(compare\_multi\_line(), "1 4")

    @patch("builtins.input", lambda: "1 2 3" )

    def test\_compare\_single\_line(self):

*self*.assertEqual(compare\_one\_line(), "1 3")

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

# Debugger

Python Debugger (PDB):  
import pdb  
pdb.set\_trace()

To debug a program with pdb:

$ python3 -m pdb my\_program.py

* help – get help
* Where – return where we are in the execution of the code
* next – will execute the next line of code
* cont – will continue running the program (F5) until we do ctrl-C and then it will break where it is.
* list – will show the code around where the debugger stopped and the current line marked with ->
* return – try to run until the current function returns.
* quit – stop the run.
* print x – will print the value of variable x

We can also add the start pdb into the source code where we want it to start:

…. Normal python code until where we think our bug is…

import pdb; pdb.set\_trace()  
 .. the rest of our python code….

Then, when we run the code from the command line:

$ python3 my\_program.py

The debugger will start automatically when it reach this line

# Packaging

Recommended site:

<http://the-hitchhikers-guide-to-packaging.readthedocs.org/en/latest/index.html>

**A Package**

A python [package](http://the-hitchhikers-guide-to-packaging.readthedocs.org/en/latest/glossary.html#term-package) is simply a directory with an \_\_init\_\_.py file inside it.

## Create the package

See the example project in <Python language>\Projects\packaging\_example

**readme.md:**

In order to be able to use markdown readme file as the package readme, you should install pypandoc package using:

>> <python root>\scripts\pip install pypandoc

See the example project for how to use it.

## Packaging

Send command:

>> python setup.py sdist

This will create a dist sub-directory in your project, and will wrap-up all of your project’s source code files into a distribution file, a compressed archive file.

### Included Files

By default, distutils does not include all files in your project’s directory in your project’s directory. Only the following files will be included by default:

* all Python source files implied by the py\_modules and packages options
* all C source files mentioned in the ext\_modules or libraries options
* scripts identified by the scripts option
* anything that looks like a test script: test/test\*.py
* Top level files named: README.txt, README, setup.py, or setup.cfg
* All files that matches the package\_data metadata. See [Installing Package Data](https://docs.python.org/2/distutils/setupscript.html#distutils-installing-package-data).
* All files that matches the data\_files metadata. See [Installing Additional Files](https://docs.python.org/2/distutils/setupscript.html#distutils-additional-files).

If you want to include additional files, then there are a couple options for including those files:

### How to include more Files

* Use Distribute – it is a package which extends Distutils with more functionality.  
  It allow you to include all files checked into your version control system.
* Write a top-level MANIFEST.in file. This is a template file which specifies which files (and file patterns) should be included. (TO-DO: link to a MANIFEST.in document)

### Before Releasing

Before releasing, it’s a good idea to double check to make sure that you have:

* The correct version number.  
  While it’s handy to append a dev marker to the version number during development, so that you can distinguish between code under development and a released version, you **never** want to publish a release with dev in the version name.
* All desired project files are included.  
  Go over the MANIFEST file, or open the archive file generated by running the **sdist** command.
* Test your distribution on a clean PC and make sure it works without any problems.

## Installing a Package

To install a package, run the following command in the package root directory:

>> python setup.py install

You can add ‘—record files.txt’ flag to get the list of all the files and where they were copied to.

## Uninstall a Package

Distutils does not include an uninstall option. Therefore, you should manually remove the files that were installed in the install process.

# Distributing Your Program

## Installing as Part of Python Libraries

Using distutils:



* The setup.py should be in the root directory of your project
* py\_modules – should include all the modules required from your project
* To install it on a new machine/venv:   
  python setup.py install
* This will install the modules as part of the python library in ‘site-packages’
* To package the installation into a .zip file:   
  $ python setup.py sdist –-format zip

This will create a ‘dist’ folder with a zip-file that contains the package that can be installed in the same way as before.

# Installing External Packages

* For numerical or scientific computing which rely on the NumPy or SciPy libraries, consider using ‘Anaconda’
* pip is the preferred packaging tool for everything else. It is included with Python 3.4 and above. Before that, you can install it.
* pip can search for packages in a central packages repository known as the Python Package Index (PyPI) and then download them and install them together with their dependencies. The PyPI nickname is the “CheesShop”.
* A “wheel” of cheese is the name of a Python binary package (which can be found and downloaded from PyPI using pip).
* https://pypi.python .org/pypi
* To upgrade pip:  
  pip install –-upgrade pip
* To install a package:  
  pip install <name of package>
* To install a local package (that was packed using distutils:

pip install <name of package .zip>

We can install it in our venv environment so we keep our main python installation on the computer clean.

* To uninstall a package:  
  pip uninstall <name of the package>

# Compile to EXE

## Using PyInstall

<https://mborgerson.com/creating-an-executable-from-a-python-script>

1. Install PyWin32
2. Install pyinstaller
3. Compile:

pyinstaller.exe --onefile -–windowed –-paths=<paths> <app.py>

If the build was successful, the final executable, **app.exe**, and any associated files, will be placed in the **dist** directory, which will be created if it doesn’t exist.

**Notes:**

* + You do not need to specify additional modules in the command as they will be automatically pulled via import statements.
  + The resulting executable will be relatively large because the Python interpreter, the application code, and all the required libraries are all packaged in (as specified by the --onefile option). Though convenient, there are some implications with this approach which you should be aware of before releasing using this method. See the [PyInstaller Manual](http://pythonhosted.org/PyInstaller/#overview-what-pyinstaller-does-and-how-it-does-it) for more information on bundling.

**Common flags:**

* --onefile is used to package everything into a single executable. If you do not specify this option, the libraries, etc. will be distributed as separate files alongside the main executable.
* --paths is used to indicate additional paths to pyinstaller to search for modules. This is required when pyInstaller doesn’t know how to find the modules in the ‘import’ clauses.  
  The path can be absolute or relative and separated by ‘;’  
  example:  
  --paths=../;C:/Python/Python27/Lib/site-packages
* --windowed prevents a console window from being displayed when the application is run. If you’re releasing a non-graphical application (i.e. a console application), you do not need to use this option.
* <app.py> the main source file of the application. The basename of this script will be used to name of the executable, however you may specify an alternative executable name using the --name option.
* --icon=<app.ico> will add <app.ico> as the icon of the executable. You can get royalty free icons from: <https://smarticons.co/>
* --version-file=version.txt – to add version information

1. To create installer

# Troubleshooting

## Files

**When I open a file, it reads spaces between every two characters!**

The file is encoded in Unicode. Open it in the following way:

import codecs

codecs.open(full\_name, encoding='utf-16')

# Python Internals

## Types

Python is a strongly typed, dynamic language.

Dynamic Typing - means that the types of the objects are only resolved in run-time. If there is a problem, the program will crash.

Strong Typing - means that python does in general **not** implicitly convert types.

For example: add(“the answer is “ , 42) will return an run-time error because python can’t convert the int to string implicitely.

## Objects

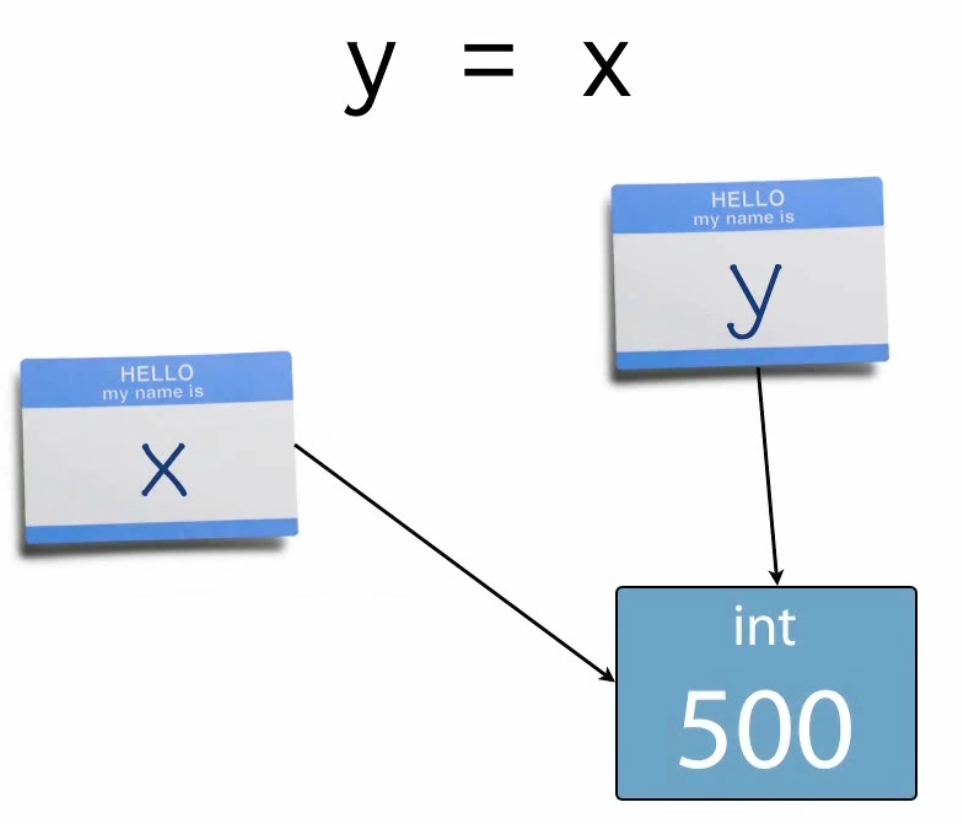
In python everything is an object! This includes all object as well as functions and modules!

## Value vs Reference Types

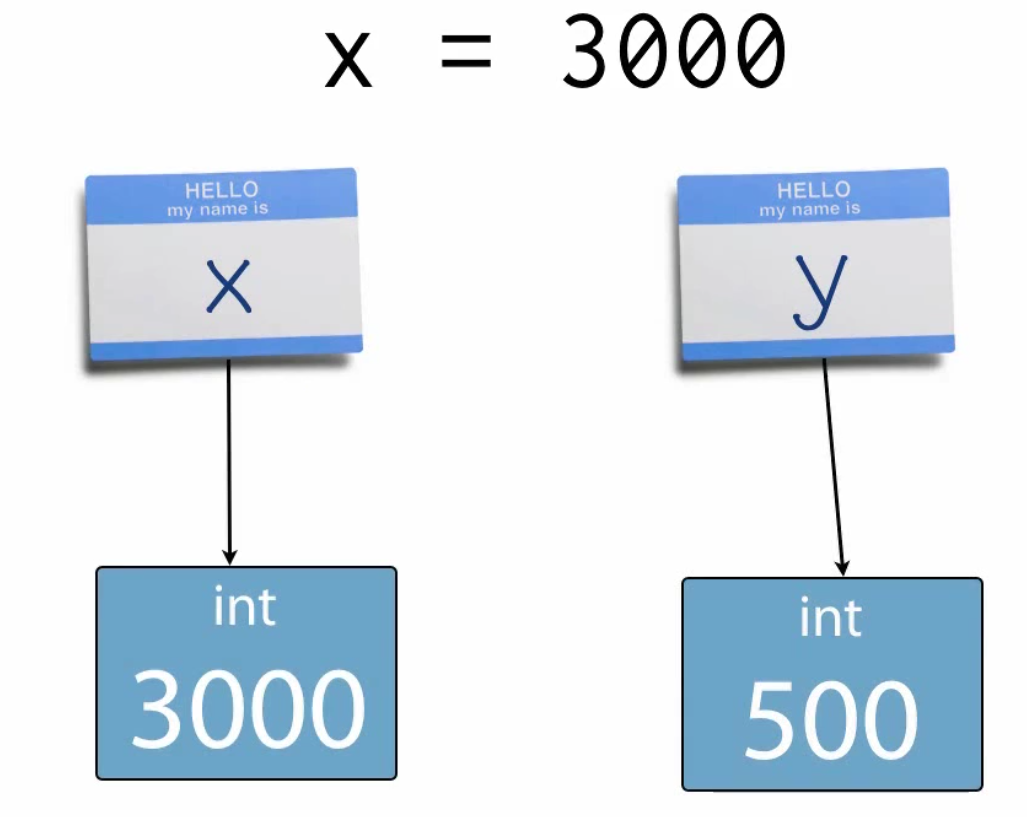
**All variables in python are reference type that points to the actual objects.**

When we assign a variable for example: x=500, python creates and int with the value of 500 and a reference x that points to the 500 int.

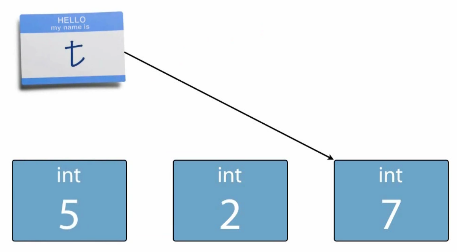
When we assign y=x, python create a reference y that points to the same int 500:



When we assign x to 3000, python creates a new int 3000 and change x to reference this new int.



t=5 # python will create an int 5 and an object t that reference this 5 int.

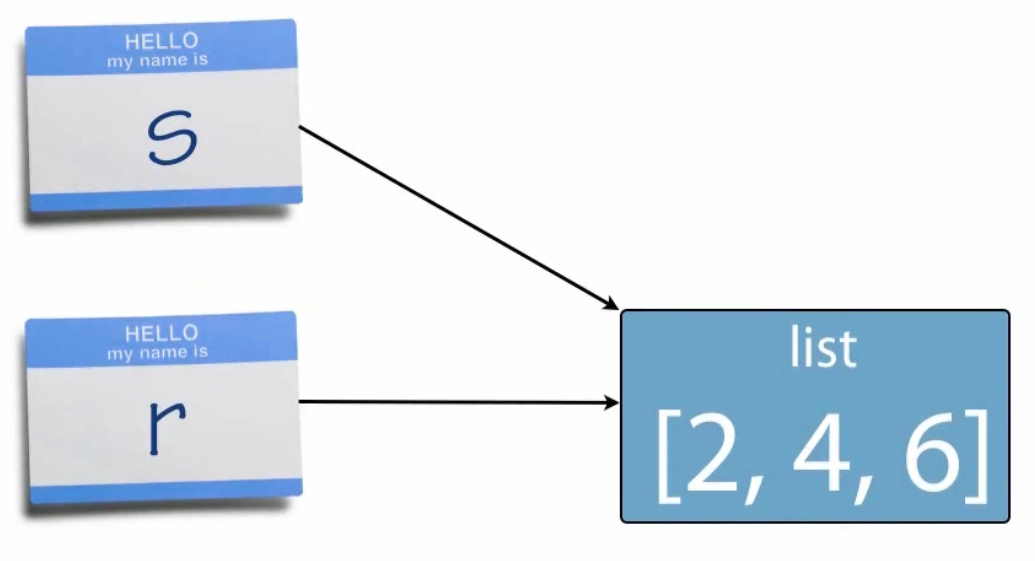
t += 2 #python will create another int 2 and will calculate 5+2. It will then create a new int object 7 and will change t to reference this new int 7. The int 5 and int 2 objects will be collected by the garbage collector:

**The assignment operator in python only ever bind objects (e.g. int 5) to names (e.g. t). It never copy any values.**

r = [2,4,6]

s = r

>> r is s  
True  
>>id(r) == id(s)  
True



## Value equality vs Identity

Value – equivalent “contents”. This can be controlled programmatically.  
To test for equivalence: use ==

Identity – same object. This is managed by the python engine and we don’t have direct control over it.  
To test for equality of identity, use is

## Function Arguments

All functions parameters are passed by reference.

## Scopes

LEGB rule:  
To resolve names, python searches from the internal namespace to the more global one until it can resolve the name and then stop, in the following order:

* **L**ocal – inside the current function
* Enclosing – Any and all enclosing functions
* Global – Top-level of module
* Built-in – Provided by the builtins module

If you need to refer to a global variable from within a function, you need to declare the variable as global:

count = 0

def set\_local\_count(new\_count):

count = new\_count

def set\_global\_count(new\_count):

global count

count = new\_count