# Python (v3.6)

*Notes open for creative commons use @ developer blog: [https://unfoldkyle.com](https://unfoldkyle.com), github: SmilingStallman, email: [kmiskell@protonmail.com](mailto:kmiskell@protonmail.com)*

**Learning Resources**
**-**Primary: Python Official Docs: [https://docs.python.org/3/tutorial/index.html](https://docs.python.org/3/tutorial/index.html)
-In conjunction with: *"Python Crash Course"* (2019)

## Intro

**About Python**
-Interpreted scripting language (no compiler needed) with OOP capabilities, excellent for automation, with large collection of micro-frameworks and extensions

-Syntax style is very minimalist (no brackets, indentations), high-level language, with variety of built in data types. Less development time than C/C++/Java.

-Modular, enables easy re-usability and easy import of existing collections, which can be used as program basis and help follow design patterns

-named after *Monty Python's Flying Circus*

**Theory**
-Beautiful code is better than ugly code
-Simple is better than complex
-Complex is better than complicated (many components better than high level of difficulty)
-Readabilty counts
-Do it the obvious way that would make most sense to programming standards, designs, etc.
-Now is better than tomorrow, which is often never. Always keep learning.

**Setup**
-python3 and python 2 come installed by default on most linux

-Will want to install pip package manager: *sudo apt-get install -y python3-pip*
-Install with: *pip3 install package_name*

-Extras: *sudo apt-get install build-essential libssl-dev libffi-dev python-dev*

**Virtual Environments**
-isolated python namespace from rest of OS, ensuring each project has own set of dependencies, etc., useful work working with diff versions, third party packages, etc.
-*pip* installed packages in 1 env are not installed in others

-Install v-environment packages: *sudo apt-get install -y python3-venv*

-Envs are directory based, with a couple scripts added to dir by *python3-venv* to set up env

-To create env, from folder want project in, run: *python3 -m venv my_env_name*
-Will generate bin, include, etc. dirs & files, similar to what *create-react-app* does with *node_modules*

-Enter env: *source my_env/bin/activate*

-Exit env: *deactivate*

### Python Interpreter

-Enter interpreter terminal via *python3* command
-Allows interactive editing and execution of code real-time

-If program contains errors, run through interpreter on terminal to get errors printed

### Basic Script

-folder/file naming convention: *my_script.py*

-Run from command line via: *python my_script.py*

### Basic Style

-Recommended 4 spaces per indentation (a tab is also 4 spaces, making them interchangeable)
-Limit lines to 79 chars max, and comments 72 chars max

-See *Python Enhancement Proposal* for more style guidelines

### ***print()***

-Takes in string objects and prints to text stream file
-Can take in multiple strings separated by commas and will print all with a space in between each one

```
print('Hello world')
print('Hello', 'world')          //same output as above
```

### Justification

*string.rjust(n)* – Justify to write by *n* spaces
*string.ljust(n)* – Justify to write by *n* spaces
*string.center(n)* – Justifies both sides, each side by *n / 2*

## Variables & Math Operators

### Variables

**-**syntax: *name = value*
  -ex. *message = "hello world"*

-note no type, no *$* to signify var, etc.
-naming convention: underscore, letters, nums only. Lowercase. Cannot start with num. *my_var*

-Variables in Python can be thought as labels: labels you can assign to values. Stored by reference.

-in interactive terminal, last printed expression result is store in temp _ variable:

```
>>> 3 * 3
9
>>> 5 + _
14
```

-Reference to undefined variable will throw error

### Comments

```
some_code    #comment

#multi-line
#comment
```

*#do actually include meaningful comments, to help with quick comprehension for others reading code*

**Multiple Assignment**
-Can assign multiple variables with multiple values in one statement with comma separation
-syntax: *x, y, z = 1, 2, 3*

-Must have value for each variable. Cannot just do *x, y, z = 0*

**Constants**
-No built in *const* variable but if want a variable to be a count give *ALL_CAPS_NAME* to indicate this to other devs (lol)

**Numbers & Math Operators**
-Typical math operands: *, / _, -, %, etc.
-Group with *( )* for order of operations

-Division with / always returns a float, even if 1.0, etc.
-Drop decimal (floor) with //            ex. *5 // 2 = 2*

-Powers: 2**3            // == 8

-Any operation with mixed type operands (ex. *float* and *int)* will result in *float*

-Round to num decimal point with global funct: *round(round_me, decimal_points)*
  -ex. *round(126.18273, 2)*              *//126.18*

-To make numbers more readable, can put underscores in difintions to represent commons. Does not change number value, but easier for human to read.
  -ex. *universe_age = 14_000_000_000*        *//universe_age === 14000000000*

## Strings

**Strings**
-Single quote pairs or double quote pairs
-Escape non-paired quote with \

-*n* – new line escape char
-*t* – adds tab to string...useful for plaintext lists and text nesting

-If wrap strings in triple single or double quote pairs (‘’’ or “”””), can span multiple lines and preserve indentations

-Concatenation: +
-Can concat with + to strings retured from expressions, stored by variables, etc.

-Strings next to each other auto concat:
  -ex. *my_string = (‘the’ ‘ string’*
                    *‘is concated’)*              *//the string is concated*

-Can only auto-concat with string literals, not strings assigned to variables, returned from expressions, etc.

-Repeat string *x* times: *x * ‘string’*
  -ex. *3 * ‘string’*      *//stringstringstring*

**String Indexing**
-Can read (only) chars in string by referencing index as if array of chars
my_string = 'Hello'
*my_string[0]* *//H*

-Access from end index back via negative nums
*my_string[-1]* //o

-Index out of bounds attempt will throw error

**String Slicing**
-Can copy out substring as if pulling range from char array
-Starting num is inclusive, end is exclusive
*my_string[2:4]* *//ll* *//chars 2, 3 as exclusive end*

-If leave end off of slice (still using :), will count missing side as start or end automatically
*my_string[:4]* *//llo*

-Can also slice using negative index
*my_string[-4:0]* *//*

-To slice to end char using negative chars use default (*my_string[-4:])*, not *0*

-Index out of bounds will use default end or start instead for out of bounds end instead of error

-As Python variables are stored by reference, strings are immutable, and thus indexes cannot be changed via *my_string[2] = 'a'*, etc.
-Variables of course, can be set to reference other strings and variable name bindings are mutable

-String Length: *len(my_string)* // 5

**String Casing Methods**
*my_string = '   this is my string   '*

-Uppercase words: *my_string.title( )* // *This Is My String*
-Uppercase whole string: *my_string.upper()* // *THIS IS MY STRING*
-lowercase whole string: *my_string.lower()* // *this is my string*

-*lower()* useful for normalization

**f-strings**
-The equivalent of { literals in JS, allowing variables to be reference by name, then output by value as part of the string. Aka. formatted string literals.

-syntax: *f"Hello my name is {name}"* *//f before quotes and reference in { }*

-Can use with any type of quote pair, including triple pairs
-*{ }* can hold variable name or expression/function output

-Python 3.6 addition. If using early version, need to use *format()* method:
*"My name is {}. This {} tale".format("Ishmael", "is my")* *//args passed in order to {}*

-Can pass in *:n* with var name to *{}* to denote string of n chars min length
-Useful for lining in strings in columns, like pseudo <*table*>
-Syntax: *{value:n}*

-Set decimal precision for num strings: *{value:.n}*

-String representation of an object: *{some_object!r}*

**Removing Whitespace**
-Removing all whitespace before records stored, compared, etc. useful for normalization

-Remove rightward whitespace: *my_string.rstrip()*
-Remove leftward whitespace: *my_string.lstrip()*
-Remove whitespace both sides: *my_string.strip()*

-As strings are immutable in python, these methods return new strings. To change ref of variable holding string: *mystring = my_strip.rstrip()*

***format()***
-Syntax: *'This {} a {}'.format('is', 'string');*

-Brackets (format fields) filled in by *format()* args in order (by default)

-Can also specify which args to use in which *format fields*
  -by *positional arg* index: *'{1}. My name is {0}'.format('Kyle', 'Hello')*
  -by *keyword arg*: *'This {food} is {taste}'.format(food='cake', taste='delicious')*
-Can combine keyword and positional. Pass in *positional* before *keyword*

-*{}* can also contain a dictionary key reference. If doing this pass *some_dict* into *format(*
  *'{some_dict[key]}'.format(dict_name)* *//no quotes around key*
-Useful in combo with *vars()*, which returns dictionary of all local variables

***repr()***
-Takes in object and returns string representation of it
-Used often to concatenate a string representation of a list, dictionary, etc.
-Can also convert numbers to string or take in expression, function call, etc.

**Strings – See also**
https://docs.python.org/3/library/stdtypes.html#textseq //text sequence strings
https://docs.python.org/3/library/stdtypes.html#string-methods //string methods
https://docs.python.org/3/reference/lexical_analysis.html#f-strings //formatted string literals
https://docs.python.org/3/library/string.html#formatstrings //format string syntax
https://docs.python.org/3/library/stdtypes.html#old-string-formatting //printf string formatting

# Lists

**Lists**
-Access via index (zero oriented), as with array. Creation:
  *my_list = [1, 4, 9, 'string', 1.92]* *//Loosely typed and can hold mix of types*

-To view list contents simply *print(my_list)*

-List index access and slicing works the same as *string* slicing (see *String Slicing* section above).
-Lists are mutable, unlike strings though:
  *my_list[3] = 'fourth'*

-List length - global function: *len(my_list)*

-Trying to access out of bounds index throws index error

-Can create nested listed
*my_nested = [[1, 8, 12], [ 'a', 'b', 'c']]*
*my_nested[1][2]   //c*

-Check if item in or not in my list via *some_val in my_list* and *some_val not in my_list*

**List Range Assignment**
-Can also set members using slice style range syntax with =
*my_list[2:4] = []   // = [] removes item*
*print(my_list)   //[1, 'string', 1.92]   //right is exclusive, as with string slice*

**List Add/Remove Methods**
-These methods, like setting range *[2:5]*, etc. directly modify the lists. No re-assignment is needed like with immutable strings. Calling *my_list.pop()* results in *my_list* having one less index.

-Append to end of list: *my_list.append( arg_x )*
-Delete item from list: *del my_list[2]   //can also del slice from iterable*

-Remove end item: *my_list.pop()*
-*pop()* returns item popped, so can *pop()* and store in variable, etc.
-Can pass index num to remove to *pop()* any index: *my_list.pop(4)   //index 4 popped*

-*del* more efficient than *pop()*, so if no need to access item on removal, use *del*

-Remove first occurrence of *value*: *my_list.remove( 'value' )*

-Append index into list, pushing existing index forward 1: *my_list.insert( index, arg_x )*
-ex. *my_list = [0, 1, 2, 3,  4]*
*my_list.insert( 2, 'hello')   //[0, 1, 'hello', 2, 3, 4]*

-Add existing iterable on list: *my_list.extend(another_list)*
-Each index in *another_list* gets own index in *my_list*

-Empty list: *my_list.clear()*

-Lists as stacks: push onto end using *.append(x)*, pop from end using *.pop()*

-Lists as queues: push onto end using *.append(x)*, pop from start using *.poplseft()*

**List Organization**
-Sort alphabetically, on list called on: *my_list.sort()*   //optional arg *reverse=True* for z to a sort

-Sort alpha temporarily (returns sorted copy): *sorted(my_list)*   //optional arg *reverse=True* for z to a

-Reverse list order called on: *my_list.reverse()*
-Produce reversed copy of list: *reversed(my_list)*

-Count num times *x* occurs in list: *my_list(x)*

-Create shallow copy of list: *my_list.copy()*

**Methods Useful with Num Lists**
-For numeric lists:

```
  min(my_list)
  max(my_list)
  sum(my_list)
```

**List Comprehensions**
-Creating a list by calling a *for* statement in the list, where each item in *for* loop is passed through *expression*, then added to list. Similar logic to *map()*

-Syntax: *my_list = [expression for item in list optional-if]*
-Equivalent to:

```
  for item in list:
      if conditional:
          expression
```

-ex. *squares = [value**2 for value in range(1,11)*

-Pretty similar to a *map()* function where *expression* is the function which JS passes all elements of iterable through, creating new iterable from results

-Can create with nested *if* and *for*. ex.
*[(x,y) for x in [1, 2, 3] for y in [3,1,4] if x != y]* is equivalent to

```
  for x in [1,2,3]:
      for y in [3,1,4]:
          if x != y:
             combs.append((x, y))          //if !=, add tuple of (x, y)
```

-As *for* loop is an expression can create nested lists via:
*[[x for x in [1,2,3,4]] for i in range(5)]* // *[[1,2,3,4], [1,2,3,4],….]*

-Note that expression is also a comprehension and that *i* is available to it (though not used here)
-Above example says, "create a full comprehension for 4 iterations, where each index (*expression)* is also a comprehension of *[1,2,3,4]*"

**Copying a List**
-Simply slice the whole list, as slicing returns a copy: *my_list[:]*

-If set new variable name to reference existing list, changing that variable with also change the existing

**Checking if Val in List**
-syntax: *value in my_list*

-Uses *in* operator to check if val in list, then returns boolean

**Tuples**
-Immutable lists
-Syntax: *my_list = (//contents)*

-Same definition as standard list except defined inside *( )* instead of *[ ]*
-Still access index via *[index]*, not *(index)*

-Trying to change index value after set will throw error
-Can change variable referencing a tuple to ref a new val, though (no constant vars in Python)

-Are iterable, so can loop through via *for*, etc.

-Empty tuple creation: *empty = ()*
-Single item tuple creation: *singleton = ("item",)* *//must have trailing comma*

## Basic Control Flow Statements

**No Brackets**
-No brackets for blocks in python. Blocks built from tabs/spaces.
-All expressions in block must have same indentation:

```
for loop #1
  for loop #2
    do this            //same indentation
    also do this       //same indentation
print(something)
```

**Comparison Operators**
*==* equals
*!=* not equal
*>, <, >=s*

-Can use *==* and *!=* to compare strings (case sensitive)

-Can compare sequences (lists, dictionaries, etc.) using *<, ==, >,* etc.
-Python checks the two comparing sequences in lexicographical ordering, where first two list items compared, then next two, etc. If *false* value returned in comparison, function returns false, else finishes returns true.

**Logical Operators**
*and*
*or*
*not*
*in* *//'some_val' in my_list* *'some_value' not in my_list*
*not*

**Booleans**
*True*
*False*

-Note that booleans must be upper cased. *true* is not a boolean.

***if* Statements**
-*if, elif, else* with no ( ), and *:* at end of 'condition line'

```
if x < 0:
  x = 0
  print('x < 0')
elif x > 0:
  print(‘x > 0’
else:
  print(‘x == 0’)
```

-*elif*, and *else* optional. Use *elif* if have multiple else conditions with *else* as final catch all, as standard.

-Can use in same way as *switch* or *case* statements in other langs, with multiple *elif* blocks, etc.

-Can check if list or tuple empty by passing list name in as if condition. Can also check for non-empty string this way:

```
        if my_list:                 //if my_list not empty
           //do stuff
```

## *for* Statements

-Iterate using *for index in iterable:* , where each loop *index* holds current index in *iterable* and loop ends when list ends

```
-Syntax:        for index in iterable:
                   index stuff to do
```

```
-ex. ints = [1, 2, 5, 9]
      for int in ints:
          print( int )      //1, 2,5,9
```

-Python <u>does not</u> have a *for($i = 0; $i < 10; $i++)* step and halting condition based *for*

-Python docs recommend not modifying an iterable while iterating over it, but instead
a) looping over copy of collection while modifying original
b) creating a new, empty collection to add state to

-As *slice* of list just returns subsection of list, can also loop through it

-Can loop through multiple iterables by wrapping iterables in *zip():*

```
   for q, a in zip(questions, answers):
       print(f‘Question: {q}, Answer: {a}’)
```

-If using *zip()* in loop on iterables of different sizes, loop will exit once smallest iterable ends

## *range()* Function

-Allows you to iterate *n* number of

```
for i in range(5)
  print(i)                //0 1 2 3 4
```

-With single are, called *n* times, with first *i* being 0

-If *print(range(10)),* object is similar to a list, but smaller. Is an *iterable* object in Python. Many Python functions can take iterables and perform an action on their members:
  -ex. *sum(range(4))*           // 6
 (
-full syntax, with optional params:   *range(start num, stop num-step, step)*
   -ex. *range(-10, -100, -30)*           //-10, -40, -70

-When using *start* and *end* args, last num will always be 1 *step* prior to the *end* num (exclusive step based end)

-Can iterate over collection (ex. *list*) by passing in *len(my_list)* to range, then interacting with *i* index of *list* for all *i* iterations produced by *range()*

- *list(range(5))* – will produce of list of nums

***break***
-*break* only breaks out of innermost loop if in nested loop

***for:   else:***
*else* in *for* syntax:

```
for x in range(5):
   //some test(s)/conditional logic
      break
else:
   //do some stuff
```

-If loop loops all the way through *range* etc., then *else* will also execute, as a *final* for the loop. If loop hits *break* before completion, loop terminates, and *else* will never execute.

-Essentially, "loop *n* number of times. If you loop all the way through *n*, then also do *else*. If you *break* and stop looping before looping *n* times, then you're done; no *else*, just exit loop.

***continue***
-Standard use. If called, following loop logic in iteration skipped, and onto next iteration

***pass***
-A placeholder. It can fufill action action, but does nothing.

***while* Statements**

```
while condition:
   body
```

-Can be exited using *break, continue* to skip to start of new iteration, and *pass* as a placeholder, as with *for* loop

-Can iterate through collection, one index per loop with    *while collection:*

-To remove all instances from list

```
while 'something' in my_list:
   my_list.remove('something')s
```

# Functions

-Use keyword *def* to define function. No *{}*. Body is indented.

-Syntax:
```
def function_name(args):
    """optional docstring"""
    //logic (body)
   return something          //optional
```

-Functions that return nothing return a *None* (a built-in name), which is generally suppressed by the interpreter
-Can return call to function, Anon function, etc.

**Docstrings**

-If have string literal as first line of function, does nothing, and acts as “documentation string” for func
-Should be a description of the function’s purpose, enclosed in triple quotes

-Can access via calling .__*doc*__ property (2 underscores both sides) of function

**Scope**
-Function execution results in a “local symbol table” being generated to hold variable assigns for function

-Variable references look first in local symbol table, then local symbol table of enclosing functions, then global symbol table, then built-in (reserved, etc.) table

-Since global, enclosing function, and current function variables all exist in diff symbol tables, can only define values for variables within their own scope (cannot set global variable from within funct)

-As scope check cascades, however, can reference global vars from within function, etc

**Args**
-args passed in stored in local symbol table of function, then referenced from there, hence *call by value*, where arg is a reference to a value

-Functions definitions are also stored in a local symbol table as a value, and thus can store functions in variable, store same function by different variable references, etc.

-Reminder: parameter is placeholder for piece of info to be passed in, arg is value when function called and val passed into it. “This function has two parameters. I pass in arguments *2* and *“three”* to it on this call.

**Positional Args**
-Standard order based based args
-If params defined in function definition, but args not passed in, error thrown

**Default Args**
-Can give parameters default values via *param_name=’value’* in definition
-If don’t pass in args to these params, default vals will be used instead

-ex. *describe_pet(pet_name='willie')*

-Default args value are defined in the scope that the function is defined in, valued at the point of definition

```
i = 5

def f(arg=i)
   print(arg)

i=6
f()      //5
```

-The default value is a mutable value. It maintains state across function calls. Ex.

```
def f(a, L=[]):
   L.append(a)
   return L

print(f(1))      //[1]
```

*print(f(2))*      *//[1, 2]*
*print(f(3))*      *//[1, 2, 3]*

-If don't want call persistent state, set to *=None* in default val definition
-Should be defined after positional params

-To make args option, use default arg set to *my_arg="* or some other existence value (ex. *null*). This allows the arg to be skipped, then value check inside function body for use or no use.

-No space between = in arg=*default* in param def

**Keyword Args**
-*key=value* pair passed into function, given a default value during function definition

-ex. *describe_pet(animal_type='hamster', pet_name='harry')*
    *//body*

-When pass in args, pass in *key* as well, which lets you pass in args in any order, even if different definition order

-ex. of call:
  *describe_pet(pet_name='harry', animal_type='hamster')*

-Should be defined after positional params, but order positional args is irrelevant

-No space between = in *key=default* in param def

**When not to Mix Keyword & Positional**
-Positional only – want param names to not be available to user, such as when names have no meaning

-Keyword only – when useful to pass in arg by key to define clear meanings

**Arbitrary Args**
-Similar to *rest* params in JS where allow to pass in any number of args to function
-Syntax: *function(*arb_args)*      *//do not reference via * (only name) inside function*
-As a general rule define them as last param.

-Inside function, arbitrary args are stored in a *tuple* (*immutable list)* and thus can be accessed via index *arb_args[some_num]*, traversed through via loops, etc.
  -ex. *print(arb_args[1])*

-Generally define as last param in method definition

-If define before other args, following args will need to be keyword (to know which are following args and which are part of arbitrary), and function should be called with arb args first:
  *my_function('argA', 'argB', *arb_args, 'some_keyword'='my arg')*

**Arbitrary Keyword Args**
-Same rules and idea as arbitrary args, but allow you to pass in keyword args, as many as desired, instead of positional style args to arb arg

-Syntax: *def my_function(positA, positB, **keyword_arbs):*      *//def*
    *//body*
  *my_function('First', 2, key_a='value', key_b=2, key_x=…)*   *//call*

-Args passed in via dictionary, so access as would standard for a dictionary

**Lambda Expressions**
-Keyword *lamba* allows to define a following small anonymous expression function
-Syntax: *lamba args: expression       //result of expression returned*

-Typically used to assign a simple expression to a variable

-ex.
```
x = lambda a, b : a + b
print( x(5, 10) )          // 15
```

-Note in definition, args seperated by *,* but not includes in *( )* and no *return* needed

-When return a lambda, are returning actual function, which can then be called, not output of function

-Useful for partial application, currying, etc. Ex.
```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)             //returns function a : a * 2
print(mydoubler(11))              //calls 11: 11 * 2, returns 22
```

**Keeping Arg State**
-Since slicing of a collection returns a copy of the collection, while not modifying the original collection, can pass in *my_collection[:]* as arg to pass in copy, instead of original.

## Dictionaries

-*key=value* data structure in Python similar to *assoc array* in PHP, *map* in JS, etc.

-Definition: *my_dict = {‘key_a’: 2, ‘keyB’: ‘someString}*
-Access Value: *my_dict[‘key_b’]*
-Add new key-value: *my_dict[‘new_key’] = ‘myValue’*

-Note that defined with *{ }* but access keys via *[ ]*

-Empty dictionary definition: *my_dict = {}*

-Remove *key* from dictionary: *del my_dict[‘my_key’]*

-Can define across multi lines as long as indentation matches:
```
favorite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby'
}
```

-Trying to access non-existent key throws error

***get()***
-Alternative way to access values:    *my_dict.get(‘keyA’, ‘return if key !exist’)*
-Second arg (optional) returned if key does not exist.

***dict()***
-allows you to create dictionaries from key value pairs
*-dict([(‘keyA’, ‘valA’), (‘keyB’, 19)]          //same syntax as, but not tuples*

**Dictionary Comprehension**
-Syntax: *{key: expression for optional-if}*
-ex. *{x: x**2 for x in (2, 4, 6)}*

-Note that like set, defined using *{}*, but unlike set, expression includes *key:*

**Dictionary Looping – key, value**
-Dictionaries are not iterable, but can get a tuple of key value pairs by calling *.items()* on dictionary.
-List returned in form:
*[('key_b', 2), ('key_b', 'the KEY')]*

-Loop Syntax: *for key, value in my_dict.items():*

-Loop Syntax alt: *for index_num, value in enumerate(my_dict):*

-*enumerate(iterable, optional_start=0)* – returns an iterable which creates a counter for iterable passed into it, allowing numeric indexing of *key-value* objects

**Dictionary Looping – keys only**
-Syntax: *for key in my_dict.keys():*
-Returns list of keys

-Can also loop through keys by calling just: *for key in my_dict:*

-As *keys()* returns a list can then call *'someKey' in my_dict.keys()* to check for existence of *'someKey'* in *my_dict*. To check for non-inclusion check *not in*.
-Since *keys()* returns a list, also can *sorted(my_dict.keys()), my_dict.keys().reverse(),* etc.

**Dictionary Looping – values only**
-Syntax: *for value in my_dict.values():*

-Like *.keys()*, returns a list of all values, so can call use for all list methods, etc.

-Includes all values, even if repeat values

**Dictionary Looping – sets**
-Sets are a python data structure of an ordered collection, allowing no duplicate values, defined and return in *{ }*

-ex. *my_set = {'python', 'javascript', 'php'}*

-Boolean if value in set: *'value' in my_set*

-*set(iterableArg)* produces a set. Can use to get unique values from any iterable including *my_list, my_dict.keys()*, *my_dict.values()* etc. as arg.

-*sets* are iterable and can be looped through
-ex. *for language in set(my_languages.values()):*

-Can create sets using comprehension, same syntax as list, except enclosed in *{}* instead of *[]*

**Lists + Dictionary Nesting**
-Lists of dictionaries useful for if want to have models of key-value pairs, one model per list index
-Can then call *len()* on to print number of models, slice to get subsection of models, etc.

-Dictionaries of lists useful for if want to have a key that has multiple values, by value being a list

-Dictionary of Dictionary useful for when want a group of key models referenced by key, which then each contain further key-value properties.
-Ex. Dictionary of users each with a dictionary as a value, holding user props

-Beware of deep many level nesting, as this significantly increases both complexity and complication of code, making harder to read, extend, etc. and makes things nightmarish for new coders on an existing system with heavily nested and many dimension data structures.

# Modules

**Intro to Modules**
-In Python, a module is simply a python .py script. Modules can be imported into another python script, and then have their functions called, variables referenced, etc. from the importing script. Pretty much the same as *include* or *require* in PHP.
-Import: *import file_name* *//note, no .py*
-Access: *file_name.some_function()*
*filename.some_var*

-When import module, elements of module entered into module symbol table, not table of importing file table, hence need to call using *file_name.whatever*

-As functions can be assigned to vars in Python, can assign module function to var, then call as far, for shorthand, if calling often
*some_function = my_module.some_function*
*some_function(args)*

-If statements/expressions exist in module, these will execute on the <u>first</u> import of a module or on execution as script

**Module Scope**

-As each module has it's own symbol table for vars, no concern for name clashes in importing vs imported files

-Can import all of module into local file (allowing direct reference without *my_module.whatever*) via:
*from my_module import **

-Can also selectively import parts of module into local symbol table via:
*from my_module import some_var, some_funct_name*

-Be wary of importing all using * as this can create name clash problems and changes the way devs expect the code to work, given Python's default module scope
-Note also * does not import names starting with _

-Can import and give alias name (for future reference) during import
*import some_module as new_name*
*from some_module import my_function as new_name*

*new_name.my_function()* *//calls*
*my_function()*

**Modules as Scripts**
-When execute module __*name*__ is set to *"__main__"*, thus can encapsulate code you only want to

run if module executed, but not imported in:
*if __name__ == "__main__":*

-Useful for testing and to provide interface to module

### Standard Modules

-Before importing module, Python checks standard Python module path for existing modules of the same name. If such a module exists, this one will be imported instead of user created module.

-Python includes nice selection of standard modules not included in core, but available in Python library for import

### Viewing Module Names

-Can get all names used in the module via globals *dir(my_module)* which returns array of strings

### Packages

-Collections of modules, where modules are part of package
-Denote via *package_name.module_name* //"dotted module name"

Add 20 mins

-Package can also have sub-packages (collection of modules within a collection of modules)

### Creating Packages

**https://towardsdatascience.com/whats-init-for-me-d70a312da583**

-Packages are defined by their containing directory. A directory *direct* is also a package *direct*.
-Inner folders inside package folder are inner packages

-In package folder file *__init__.py* exists. This contains import statements for all files and sub-packages to import into package

### *__init.py__* Imports

-Can import into *__init.py__* in multiple ways

I) Import all into one package namespace
*from .some_module import **
*from .inner_pack import **

-Call after import *my_package.some_method()*

-This imports all into *outermost_package* namespace and all names can be referenced via *outer_most_package.name*, regardless of in inner package

-Benefits: All names in all modules accessible from calling main package

-Disadvantages: Nameclash potentials and slowdown from large namespace. Need to name all user hidden names with _ start.

-Useful with packages where users switching modules a lot, and not many modules. Make sure names properly unique and very descriptive.s

II) Import modules into separate namespaces within package
*import my_package.some_module*

*import my_package.inner_pack.some_module*

-Call after import *my_package.inner_pack.some_module.some_method()*

-Note that inner package *__init__.py* also require full package path in their imports
-To import without having to call package name, *from my_package import my_module, inner-packs*

-Can reduce call length by importing modules/inner packages individually using *from* import
-This gives each imported package/module its own namespace
*from my_package import inner_pack, base_module*

*base_module.print_me()*
*inner_pack.inner_module.print_me()*

**Relative Imports**
-If importing in sub-package, can import using sub-dir dots before import name to import from higher level, current level, etc. folders

*from .. import higher_module*

# I/O

**Strings**
-See *Strings* section on notes page 03 for string formatting

**Files - *open()***
-Create file object from file on web server via *open()* method. Then, once opened, can call various methods on object to read, write, etc

-Open file: *var_name = open('filename', mode)*

Modes
*r* – read only *w* – write only *a* – append to end of file *r+* - read & write

-By default, files open in *text mode,* with strings for r/w to file. Can also specify different encodings via optional *encoding* arg passed in to *open()*

-By default, \n, etc. out-converted to match matching platform encoding (Mac, Win, etc.) for special char

-Useful to open file using the *with* keyword, as this way file will close even if exception hit:
*with open('my_file', 'r') as f:*
*read_data = f.read()*
-If not enclosing in *with* make sure to cal *file_var.close()* to free sys resources (garbage collector will also eventually clean also)

-To read in *byte* mode, add *b* to end of mode or *rb+*

**File Methods**

-Python reads/writes through files using a pointer. Reading to end of file keeps pointer at end. File methods read/write from last char last method call left pointer at.

- *read(size)* - reads entire file by default
-If pass in *size* will only read up to *size* chars (in text mode) or *size* bytes (binary mode)

- *readline()* - reads single line at a time. Python appends a *'\n'* to each line returned, except end of file, which returns *''*. Blank lines are returned as *'\n'*.

-Can loop over lines via *for line in file_var:*

-Can get list of all file lines (one line per index) via *list(file_var)* or *file_var.readlines()*
-Note that these methods will move the pointer to end of file

-To move file pointer, call *file_var.seek(n, opt-whence)* where *n* is the char/byte to move to
-optional *whence* sets point place to move *n* from, with values *0* for start of file (default), *1* for current pointer position, *2* for end of file

-Ex.

```
  file_var.seek(0)            //go to first char in file
  file_var.seek(-3, 2)        //go to third char before file end
```

-Note that in text files, only seeking relative to beginning of file, and move to end of file via *seek(0, 2)* are allowed and any other seek will throw error

- *file_var.write(write_me)* writes to file, from current pointer position. This means if 150 chars into file, writing 50 chars will overwrite chars 151-200.
-Can take in a *string* for text files or bytes object if in *binary* mode

-If writing non-string to text file, must first convert with *repr(object)* or *str(object)*

-Additional less used file methods like *isatty()* and *truncate()* available

**User Input**
- *input(string_prompt)* pauses the program when hit and displays *string_prompts*, then continues running after user has input data. Returns input entered by user as string.

-To cast input to int *int(input)*

# Exceptions

**Syntax Errors**
-Syntax errors will halt program at error point at display message, traceback, etc. showing line where error occurred, and info on error

**Exception Basics**
-Errors that occur during execution throw exceptions. Unhandled exceptions will cause the program to halt.

-Instead of try-catch, Python uses try-except. Syntax:

```
  try:
    //try this logic
  except Exception:
     //exception has occurred, so do this
```

-*Exception* is the base exception object in Python, but can create own exception type, or use other built in types, such as *OSError* or *ZeroDivisionError*

-*try* can be followed with multiple *exception* statements, similar to an *elseif*, where each has a different *exception* type (ex. *ZeroDivsionError)*. This allows handling of multiple exception types in same *try* via diff logic for each exception type.

-Can also create *except* check that triggers for multiple exception types via:
*except (RuntimeError, TypeError, NameError):*

-Can mimic *if: elif: elif: else:* via
*try:*
*except someType:*
*except anotherType:*
*except:*
where last *except* catches all other exceptions, sans ones above it

-Can also follow *try: //body* with *else: //body*. Here, *else* will execute if and only if no exception is throw

**Exception Access**
-To access exception from within handling logic, assign exception to alias via
*except Exception as excep_name:*

-Exception object contains a variety of properties, such as *type(excep_name)* and *excep_name.args*

-If *print() my_exception*, will print error message tied to exception

**Raising Exception**
-Can manually throw a specified exception with *raise* statement
-Syntax: *raise NameError('Bad Name')* *//throws exception with NameError: HiThere*

**User Defined Exceptions**
THIS SECTION REQUIRES CLASSES. COME BACK TO AFTER LEARN PYTHON CLASSES

# Classes

-Python is fully OOP featured, with classes, inheritance, overriding, multiple inheritance, extension, etc. with classes created at runtime, allowing runtime modification after creation

-Multiple names can be bound to same object instance

Additional Info from below: https://python-textbok.readthedocs.io/en/1.0/Variables_and_Scope.html

**Namespaces**
-Names – refer to value in memory. A variable. All names referencing the same value, reference the same place in memory.

-Namespace - a set of names mapped to a set of values. Namespaces are separate from each other, allowing same name definitions (ex. *my_function*) in the same module, as long as names exist in diff namespaces. Prevent name clashes.

-Local namespace – Function level. Each function gets own namespace, created at call, and cleared when function finishes. Recursive functions get own namespace for each recursive call, etc.

-Global namespace – lowest level module namespace, outside of functions, etc.. Individual global namespace for each module. Built when module loaded, until program end.

-Built-in namespace – holds names for all built-in functions and exceptions. Built when Python interpreter starts, until Python exits.

**Scope**

-Scope – where a variable is accessible in the code

-Lifetime – the duration in which the variable exists. Determined in part by scope.

-Local scope – innermost scope, which has namespace of current function

-Enclosing-function scope – names defined in the local scope of any and all enclosing functions

-Global – module level scope, for names defined at root of module

-Built in scope - holds built in names

-Can remember Python scope via LEGB

```
//global scope

  def outer():
     //enclosing scope for inner
     //local scope for outer
     def inner():
        //local scope for inner
```

-Python scope resolution (trying to find name) is relative to the current location. First it looks if name exists in local scope, then cascades up through enclosing (outer) functions, then checks global, then finally built in

-ex. If *x* is defined in a function defined in global and *x* is also defined in global, and the function prints *x,* it will print the *x* from the function namespace, not the global namespace.

-ex. If *y* is defined in global, and *y* is defined in a function defined in global, and that function contains and inner function which also defines *y*, then that innermost function prints *y*, it will print the *y* defined inside of it. If the innermost *y* was not defined, the innermost function would print the enclosing (outer) function's *y*

-While scope resolution for reference is cascading, searching for name in LEGB order, assignment is not cascading, and a variable with the same name created in an inner scope will not overwrite the existing variable in the outer scope. Same applies when using *del*.

-ex.

```
  //global
  x = 10

  def my_func():
     x = 2
     print(x)          //2

  print(x)             //10
```

-Note that Python does not implement block level scope, meaning for loops, if-else, etc. do not have their own separate scope

-Scopes are determined textually. The scope of a function within the module is in the global scope with the name of the module name. This name is also shared by the namespace.

**Attributes**
-Python docs define all members of an object as an attribute (both behaviors and properties)

-Access via *my_object.attribute*

-Can delete attributes via *del my_object.attribute*
-Be careful with this, as this could create non-uniform objects and essentially alter the object to something different that what it was constructed to be

-*del* of attribute from object removes binding of attr name from namespace

***global* variables**
-Declaring a variable with the *global* keyword will tell Python to use variable name from global scope. Allows you to change global variable value from inside function(s).

-Also allows to create variable with lifetime that exists outside of function, as variable does not need to exist prior, and instead can be created inside function, then will still exist in global even after function call.

-*syntax: global my_var*
*my_var = 10*

***nonlocal* variables**
-Nonlocal is another keyword similar to *global* but instead of placing name in global namespace, is used mostly inside nested functions

-When define variable as *nonlocal* variable exists in nearest enclosing function. For example, if global has function, nested in function, nested in function, where each function defines *x,* if set innermost *x* as *nonlocal* innermost *x* is now same namespace as middle function *x*. Changing inner function *x* val thus also changes middle function *x* value. Outermost function *x* does not change.

-nonlocal variables allow modification of enclosing function variables from within inner functions

**OFFICIAL COMPLETE – 1-7 (resume at 8, at "**The use of the `else` clause is better than adding**")**
**CRASH COURSE COMPLETE – 1-8**