LEARN A BIT LEAFLET

PYTHON REFERENCE

DATA TYPES, VARIABLES

```
# EXAMPLE                   # TYPE
# CASTING
var_name = 5                # int               int()
var_name = 5.0              # float (decimal)   float()
var_name = "hi"             # string            str()
var_name = True / False     # boolean           bool()
var_name = [1,2]            # list              list()
var_name = {1,2}            # set               set()
var_name = (1,2)            # tuple             tuple()
var_name = {1:2}            # dictionary        dict()
```

STRINGS

```
# FUNCTION                  # NOTES
str.upper()                 # upper case all
str.lower()                 # lower case all
str.title()                 # title case all words
str.capitalize()            # capitalize first word only

str.count("")
str.find("")                # return first index, else -1

str.split(delimiter)        # return list of strings
str.replace("", "")         # replace all first with second
str.strip()                 # remove whitespace from ends
str.rstrip()                # remove whitespace from right
str.lstrip()                # remove whitespace from left
str.splitlines()            # return list based on \n

str.startswith("")          # case sensitive
str.isalnum()               # is string letters/digits
str.isalpha()               # is the string only letters
str.isdigit()               # is string only digits
str.isupper()               # is string all upper case
str.islower()               # is string all lower case
```

LISTS/ARRAYS

```
lst = []

# function                # notes
lst.index(item)
lst.count(item)
lst.pop()                 # index OR default = last index
lst.copy()

lst.append(item)
lst.insert(item, index)   # not replacing curr index item
lst.extend(iteranle)      # iterable (add each item)
lst.remove(item)
lst.reverse()
lst.sort()                # -1 decr. OR default = incr.
lst[start:end:step]       # end is exclusive
```

SETS

```
set1 = {1, 2, 3}
set2 = {1, 2.0, True}

# function              # notes
set.add(item)           # add 1 item, can be iter
set.update(iterable)    # add each item in iterable
set.remove(item)        # if not in set, err
set.discard(item)       # if not in set, no err
set.pop()               # remove random item

set.intersection(set2)
set.difference(set2)
set.union(set2)

set1.issuperset(set2)   # T/F: is set2 superset of s1
set1.issubset(set2)     # T/F: is set2 subset of s1
set1.isdisjoint(set2)
```

TUPLES

```
tuple = (1, 2, 3)

# function          # notes
tuple.index(item)   # index of first occurrance
tuple.count(item)
```

DICTIONARIES

```
dict = {'key1' : 'val1', 'key2' : 'val2'}
# key can not be a mutable type (no lists)

# function                # notes (k:key, v:value)
dict[k] = v               # redo OR add 1 k/v
dict.update({k:v, k:v})   # redo OR add 1+ k/v

dict.pop(k)               # remove: if k None throw err
dict.popitem()            # remove: pop() most recent k/v

dict[k]                   # get v: if k None throw err
dict.get(k, def)          # get v: if k None ret def v
dict.setdefault(k, def)   # get v: if k None insert def v

dict.keys()
dict.values()
dict.copy()
```

OPERATORS

```
# ARITHMETIC OPERATORS
+           # addition
-           # subtraction
*           # multiplication
/           # division (decimal result)
//          # division (round down to int)
%           #  modulo (remainder only)
**          # exponentation (to the power)
```

OPERATORS

```
# BITWISE OPERATORS
&           # bitwise AND between two numbers
|           # bitwise OR between two numbers
^           # bitwise XOR between two numbers
~           # invert all bits (of number in binary)
>>          # bitwise shift right
<<          # bitwise shift left (zero fill)
```

CONDITIONALS

Conditionals are also called If-Statements or Branching.

```
if condition:
    # execute if this first condition is True
else if condition:
    # execute if only this condition is True
else:
    # execute if all conditions fail


# CONDITION: COMPARISON OPERATORS
==        # equal to
!=        # not equal to
<=        # less than or equal to
>=        # greater than or equal to
<         # less than
>         # greater than


# CONDITION COMBINATION: LOGICAL OPERATORS
and &&    # ret True if both conditions are True
or  ||    # ret True if at least one condition is True
not       # ret True if value is not False
is        # ret True if both values are same object
in        # ret True if a sequence is found in object
```

LOOPS

```
# WHILE LOOP
while condition:
    # execute while this condition is True

# FOR LOOP VARIANTS: LIST, STRING, RANGE OF NUMBERS
mylist = []
for item in mylist:
    # execute line
    # you can rename 'item' to whatever you want

string = "hello"
for letter in string:
    # execute line
    # you can rename 'letter' to whatever you want

for number in range(start, end (exclusive)):
    # execute line
    # range begins at number 0 by default
    # you can rename 'number' to whatever you want


# STATEMENTS
break        # stop the loop entirely
continue     # move onto next iteration in the loop
```

CLASSES

```
# Parameter = placeholder name for a value to be passed
# Argument = the actual value passed to a function

class ClassName:
    def __init__(self, value1, value2):
        self.var1 = value1
        self.var2 = value2

    # specify default value for argument
    def func1(self, value1 = default):
        return value1

    # specify data type for argument
    def func2(self, value1 : int):
        return value1

    # specify what type to return (optional)
    def func3(self) -> int:
        return 1

    # call other functions and values inside a function
    def function4(self):
        r = self.func1()            # value1 = default
        r = self.func1(1)           # value1 = 1
        r = self.func1(value1 = 1)  # value1 = 1
```

OBJECTS

```
my_object = ClassName(arg_one, arg_two)
my_object.var1 = "some value"
my_object.var1 = "some value"
my_object.func1("some value")

del my_object
```

CLASSES WITH INHERITANCE

```
# PARENT CLASSES
class ParentClass1:
    def __init__(self, value1):
        self.parent_var1 = value1

    def parent_func1(self, value1 = default):
        return value1

class ParentClass2:
    def __init__(self, value1):
        self.parent_var1 = value1

# CHILD CLASS
# you can inherit more than one class at a time!
class ChildClass(ParentClass1, ParentClass2):
    def __init__(self, value1, value2, value3):
        ParentClass1.__init__(self, value1)
        ParentClass2.__init__(self, value2)
        self.child_var1 = value3

    def child_func1(self, value1 = default):
        return value1
```

OBJECTS WITH INHERITANCE

```
my_object = ChildClass(arg1, arg2, arg3)
my_object.parent_var1
my_object.child_var1
my_object.parent_func1(arg1)
my_object.child_func1(arg1)

del my_object
```

POLYMORPHISM FUNCTIONS

```
class Animal:
  def __init__(self, species, name):
    self.species = species
    self.name = name

  def noise(self):
    return "Woof!"

# an empty class that inherits data/funcs from Animal
class Dog(Animal):
  pass

# this class overrides the noise class from Animal
class Cat(Animal):
  def noise(self):
    return "Meow!"

# polymorphism runs same method on different classes
dog = Dog("Golden Retriever", "Rusty")
cat = Cat("Mainecoon", "Rubert")
for animal in (dog, cat):
  animal.move()
```

LAMBDA FUNCTIONS

```
func_name = lambda arg1, arg2 : expression


# VARIANT 1: STANDALONE LAMBDA
func = lambda arg1, arg2 : arg1 * arg2
func(1, 2)


# VARIANT 2: COMPOUNDED FUNCTIONS
def func(arg1):
  return lambda arg2 : arg1 + arg2

supplied_1st_arg = func(1)                # arg1
supplied_2nd_arg = supplied_1st_arg(2)    # arg2
```

HASHSETS

```
# VARIANT 1: PYTHON SETS
hashset = {}

hashset.add(1)
hashmap.add(1)      # will not be duplicated
hashmap.add(2)
hashmap.pop(1)
```

HASHMAP

```
# VARIANT 1: PYTHON DICTIONARIES
hashmap = {}
hashmap["key"] = "value"
hashmap.pop("key")

# Hash Func Type    # Example
# Mod               # h(k) = k % m
# Multiplication    # h(k) = k % m
# Universal         # h(k) = ((a*k+b) % p) % m

# k = number representing key, generated from hash()
# a, b = random constants
# m = prime number
```

STACK

```
# VARIANT 1: PYTHON LISTS
stack = []

# push
stack.append('a')
stack.append('b')

# pop
stack.pop() # removes 'b'
```

QUEUE

```
# VARIANT 1: PYTHON LISTS
# built-in, a little slow with push/pops
queue = []
queue.append('a')   # push
queue.append('b')   # push
queue.pop(0)        # pop, removes 'a'
```

```
# VARIANT 2: DEQUE
# import, can be infinite, quicker push/pops O(1)
from collections import deque
queue = deque()
queue.append('a')   # push
queue.append('b')   # push
queue.popleft()     # pop, removes 'a'
```

```
# VARIANT 3: QUEUE
# import, not infinite size, quicker push/pops O(1)
from queue import Queue
queue = Queue(maxsize = num)
queue.put('a')      # push
queue.put('b')      # push
queue.get()         # pop, removes 'a'
queue.full()        # T/F check
queue.empty()       # T/F check
```