CREATED BY LIAM GIBBINGS

FUNCTIONS, METHODS & ATTRIBUTES

NUMBERS

- cmath.*pi* and cmath.*e*: Returns values of both pi and exponential const. respectively.
- math.*sqrt*(integer1): Returns square root of integer, use cmath.*sqrt* for negatives.
- math.floor(integer1): Returns integer rounded **down** to nearest whole.
- math.*ceil*(integer1): Returns integer rounded **up** to nearest whole.
- *round*(integer1, [NumOfDigits]): Rounds integer depending on number of digits specified, default is to nearest whole.
- *range*(integer1, integer2, [integer3]): Generates a list of integers starting from integer1 up to but not including integer2. Integer3 specifies step over amount.

STRINGS

- 'string1'.*find*('string2'): Returns index of string2 string in referenced string1.
- 'string1'.*join*(list1): Returns string value of list1 but with string1 in between words.
- 'string1'.*lower/upper*(): Returns completely lower/upper case version of string1.
- 'string1'.*strip():* Returns one string with whitespace removed.
- 'string1'.*split*(['string2']): Returns list with string2 removed from string1. If string2 is not specified, white space is assumed.
- 'string1'.*title*(): Returns string with all first letters capitalized.
- 'string1'.*replace*('string2','string3'): returns string1 but with all instances of string2 replaced with string 3.
- 'string1'.*count*('string2'): Returns the number of times that string2 appears in string1.
- 'string1'.*endswith*('string2'): Returns *True* or *False* value if string1 ends with string2. Use 'string1'.*startswith*('string2') for the reverse.
- 'string1'.*count*(object1): Returns the number of times that object1 appears in string1.

LISTS

- In a list (just like a string) you can index a certain point in the list from index 0 to the end using square brackets, you can use negative indexes to reference from the end of a list. You can also use this method to change values in a list individually: list1[0] = 'string1'. You can use del to delete entries.
- Just like indexing, you can also slice lists using '**:**' inside the square brackets: list1[0:8]. You don't have to specify either value, list1[5:] or list1[-5:] would reference entries 5 onwards. You can use slicing to assign values to multiple positions at once. You can also use list1[:] to create a copy of the list without affecting the original.

- list1.*append*(object1): Adds object1 to the end of list1.
- list1.*count*(object1): Returns the number of times that object1 appears in list1.
- list1.extend(object1): Extends list1 with object1 creating a whole list instead of append which jams a an object inside the list.
- list1.*index*(object1): Returns the position from 0 where object1 is located in list1.
- list1.*insert*(index1, object2): Inserts object2 at index1.
- list1.*pop*([index1]): Deletes index1 and returns it, default is end of list1.
- list1.*remove*(object1): Removes the first instance of object1 from list1.
- list1.*reverse*(): Reverses the order of list1.
- list1.*sort*(): Sorts a list from lowest value to highest, both numbers and letters. Supply "reverse=True" as a parameter for reverse sorting. Supply "key=function1" as a parameter to sort according to a function that returns a number, for example "key=len" would sort the list according to length.
- *zip*(list1, list2, list3,…): Puts together several lists returning a new list with several tuples.
- The *sorted*(object1) and *reversed*(object1) functions do the same as reverse and sort but return a value instead of just changing the object.

DICTIONARIES

- Dictionaries are declared in curley {} braces, in the form {KEY:VALUE,KEY:VALUE} and can be addressed using the key using dictionary1[KEY], this can also be used to input entries to the dictionary using dictionary1[NEWKEY] = VALUE.
- dictionary1.*clear*(): Clears contents of a source dictionary, all variables referring to dictionary1 become empty dictionaries rather than still referring to the source as with x={}.
- dictionary1.*copy*(): Returns exact copy of dictionary1.
- dictionary1.*get*(KEY1, [object1]): Returns value of key1 if key1 exists, returns object1 if not. If object1 is not specified, default is *None*.
- dictionary1.*setdefault*(KEY1, [object1]): Returns value of key1 if key1 exists, returns object1 if not. If object1 is not specified, default is *None*. Also creates entries into dictionary1 if key doesn't exist using key1 and object1 as key and value respectively.
- dictionary1.*has_key*(KEY1): Returns True or False value for whether dictionary1 holds KEY1.
- dictionary1.*items*(): Returns each key and value in dictionary1 held in a separate tuple, and all tuples held in a list. Use .*iteritems*() for an iterator version. Use .*keys*() and .*iterkeys*() for keys only. Use .*values*() and .*itervalues*() for values only.
- dictionary1.*pop*(KEY1): Deletes key and value of key1 and returns it as a tuple.
- dictionary1.*popitem*(): Deletes leftmost entry in dictionary1 and returns it. (As I understand)
- dictionary1.*update*(dictionary2): updates dictionary1 with values from identical keys in dictionary2.

CREATED BY LIAM GIBBINGS

ABSTRACTION

FUNCTIONS

- Functions are defined using def *NAME(parameter1,parameter2). They don't have to return a value (in Python only), to return a value use return *VALUE. to simply break of out the function, use the return statement on its own.
- You can specify the parameter name of a certain parameter by using *PARAMETERNAME=*VALUE so as to avoid confusion with several parameters, you can also use the same format when declaring the function to give parameters a default value.
- You can allow a function to feed in more than one of 1 parameter by putting an asterisk in front of the parameter declaration, these will be stored as a tuple. To allow specifying the parameter names of more than one parameter, you can use a double asterisk, these will be stored as a dictionary.
- when declaring functions, variables used inside the function will be local even if a global is specified with the same name, to tell python to modify the global variable instead of creating a local, use the global *VARIABLE statement to tell it to modify the globally used version of the variable name.

OBJECT ORIENTED PROGRAMMING

- **Polymorphism:** The ability to perform operations with no prior knowledge of the data types.
- **Encapsulation:** The principle of hiding unnecessary detail.
- **Inheritance:** The method of making similar subclasses from properties of a superclass inheriting its methods and attributes.
- Classes are defined using class example:
    - Methods are defined as functions would in the global namespace. When defining a method, using *self* as the first parameter passes the object you are modifying into the method, this is so that the object can actually be modified.
    - Attributes are also defined using self; in the class 'Person', the attribute 'age' would be declared as *self*.age = n.
    - To define a subclass of a class, when defining it, put the superclass in parentheses, for example class Bird(Animal): would make the subclass 'Bird' from its superclass 'Animal'.
    - You can overwrite methods in a subclass by redefining them.
- *isinstance*(object1, TYPE1): Returns a Boolean for if object one is type1, also works with classes.
- *Issubclass*(sub, super): Returns a Boolean for if sub is a subclass of super.

MAGIC METHODS

- __init__(self): A method that is called automatically when an object has been created.
- __getitem__ (self, key): A method that adds indexing support for objects.

- __setitem__(self, key, value): adds support for setting values using indexing.
- __delitem__(self, key): Adds support for deleting entries using indexing.
- __getattr__(self, name): Automatically called when the attribute name is accessed and the object has no such attribute.
- __setattr__(self, name , value): Auto called when attempt to bind name to value is made.
- __delattr__(self, name): Auto called when attempt to delete name is made.
- __iter__(self):
    - The __iter__ method itself should execute: return *self.* Only because the actual __iter__ means that the objects will be iterators
    - Define a method called *next*(self) which returns *self*.value after determining what the value will be.
    - Using a generator is probably better than __iter__.

EXCEPTIONS

- Use the raise statement to raise an error you can raise different exceptions including *Exception, IOException* etc. Different exception types are available in module exceptions.
- Use the try statement to execute a block of code that could potentially result in an error, then use except *EXCEPTIONTYPE, to execute a block of code provided that error type has occurred. You can add as many except statements as you want to account for many exception types or you could use one except with exception types enclosed in brackets and separated by commas. To catch all errors regardless of type, don't use any arguments.
- You can get access to the error message itself by using Exception as e where e is an exception object.
- You can add an else clause to the exception to execute code in the event that there are no errors, **this is very useful for looping over a block of code without if statements using while True.**
- You can use the finally clause to clean up, such as closing files or network sockets, this statement is executed regardless of the program catching an error or not.

FILE HANDLING

- To open a file and store it as an addressable and iterable object use *open*(path, [mode]) where path is where the file is stored (use *.getcwd*() from the os module to get directory of running program). As the file is an object, it must be assigned to a variable.
- Mode is how the file modified, if at all:
    - **r**: read mode (use r+ for reading and writing.)
    - **w**: write mode (overwrite entire file, file will be created if it doesn't exist when writing or appending)
    - **a**: append mode (write to end of file on new line)
    - **b**: binary mode (must be used with another mode)

- file1.*read*([int]): reads int number of characters into file1 or whole file if not specified.
- file1.*write*('string1'): writes string 1 to file, can write new lines using \n for a line break.
- file1.*close*(): closes file1, should always close file when finished using.
- file1.*seek*(int): moves current position to integer enabling writing to specific positions.
- file1.*tell*(): returns current position in file.
- file1.*readline*([int]): returns next line up until int number of chars, if int not specified returns entire line.
- file1.*readlines*(): returns each line in a list as a separate item thus indexing possible.
- file1.*writelines*(list1): writes each list item to file1, must specify new lines using \n
- You can use the with statement to close files when the block closes instead of using file.close():
    - with open('file', 'mode') as filevar:
        filevar.method()
- After opening a file you can iterate over it in a for loop where the variable will be each line.

THE STANDARD LIBRARY

OS

- .*system*('string1'): executes string1 as if input to the command line.
- .*getcwd*(): returns current working directory as string.
- .*path.**: .path has some interesting functions for manipulating and reading directories.

WEBBROWSER

- .*open*('string1'): opens string1 in default browser.

TIME

- In python, time is represented as a tuple:
    - (year,month,dayofmonth,hour,minute,second,dayofweek,DSTbool)
- .*asctime*([tuple1]): converts tuple1 to a string, if no parameters, prints current time.
- .*sleep*(int): puts program to sleep for int number of seconds.
- .*strptime*('string1'): converts string1 to a time tuple.

RANDOM

- .*random*(): returns a random floating point number between 0 and 1.
- .*getrandbits*(int): returns a random number that uses int bits.
- .*uniform*(int1,int2): returns a random floating point number between int1 and int2.
- .*randrange*([start], stop, [step]): returns a random integer contained in the range.
- .*shuffle*(list1): randomizes order of elements in list1.

RE

- Regular expressions are for searching for patterns in strings, the use of pattern formatting can be a useful tool for search utilities etc.
- Regular expressions can be used to match more than one string to a pattern or specify alternatives or even making optional parts to the pattern.
- a wildcard is where the symbol can represent anything for example 'li.m' would satisfy 'liam' and 'lism'.
- to specify alternatives use |. For example 'python|perl' would satisfy both 'python' and 'perl'.
- adding a question mark after a subpattern (using parentheses) makes the pattern optional. For example '(www.)?python.org' would satisfy both 'www.python.org' and 'python.org'.
- .*compile*('string1'): returns an object for the pattern to avoid keep typing long complex patterns.
- .*search*(pattern,'string1'): returns a Boolean value for if the pattern was found in string1.
- .*split*(pattern,'string1'): splits to string to a list for every occurrence of the pattern and returns it.

NETWORKS

SOCKET (BASICS)

- For a server, declare the socket object: s = socket.*socket*()
- Bind the servers host and port to socket: s.*bind*((host,port))
    - use socket.*gethostname*() to get current local host.
- Start listening for connections where int is allowed number of queued client : s.*listen*(int)
- loop infinitely to accept connections:
    - while True:

```
c, addr = s.accept() #  returns client as obj and client address as str
c.send('send this string back to client')
c.close() # always close the socket just as with files.
```

- for a client, declare socket object: s = socket.*socket*()
- connect to servers host and port (port forwarding required for internet): s.*connect*((host, port))
- s.*recv*(int): receive data from server as int number of bytes (usually 1024)