**Python Glossary**

**Class**

Python is an Language that supports the Object Oriented Programming paradigm. Like other OOP languages, Python has classes which are defined wireframes of objects. Python supports class inheritance. A class may have many subclasses but may only inherit directly from one superclass.

Syntax

class ClassName(object):

"""This is a class"""

class\_variable

def \_\_init\_\_(self,\*args):

self.args = args

def \_\_repr\_\_(self):

return "Something to represent the object as a string"

def other\_method(self,\*args):

# do something else

Example

class Horse(object):

"""Horse represents a Horse"""

species = "Equus ferus caballus"

def \_\_init\_\_(self,color,weight,wild=False):

self.color = color

self.weight = weight

self.wild = wild

def \_\_repr\_\_(self):

return "%s horse weighing %f and wild status is %b" % (self.color,self.weight,self.wild)

def make\_sound(self):

print "neighhhh"

def movement(self):

return "walk"

Syntax

class ClassName(SuperClass):

# same as above

# use 'super' keyword to get from above

Example

class RaceHorse(Horse):

"""A faster horse that inherits from Horse"""

def movement(self):

return "run"

def movement\_slow(self):

return super(Horse,self).movement()

def \_\_repr\_\_(self):

return "%s race horse weighing %f and wild status is %b" (self.color,self.weight,self.wild)

>> horse3 = RaceHorse("white",200)

>> print horse3.movement\_slow()

"walk"

>> print horse3.movement()

"run"

**Comments**

**Single-line Comments**

Augmenting code with human readable descriptions can help document design decisions.

Example

# this is a single line comment.

**Multi-line Comments**

Some comments need to span several lines, use this if you have more than 4 single line comments in a row.

Example

'''

this is

a multi-line

comment, i am handy for commenting out whole

chunks of code very fast

'''

**Dictionaries**

Dictionaries are Python's built-in associative data type. A dictionary is made of key-value pairs where each key corresponds to a value. Like sets, dictionaries are unordered. A few notes about keys and values: \* The key must be immutable and hashable while the value can be of any type. Common examples of keys are tuples, strings and numbers. \* A single dictionary can contain keys of varying types and values of varying types.

Syntax

dict() #creates new empty dictionary

{} #creates new empty dictionary

Example

>> my\_dict = {}

>> content\_of\_value1 = "abcd"

>> content\_of\_value2 = "wxyz"

>> my\_dict.update({"key\_name1":content\_of\_value1})

>> my\_dict.update({"key\_name2":content\_of\_value2})

>> my\_dict

{'key\_name1':"abcd", 'key\_name2':"wxyz"}

>> my\_dict.get("key\_name2")

"wxyz"

Syntax

{key1:value1,key2:value2}

Example

>> my\_dict = {"key1":[1,2,3],"key2":"I am a string",123:456}

>> my\_dict["key1"] #[1,2,3]

>> my\_dict[123] #456

>> my\_dict["new key"] = "New value"

>> print my\_dict

{"key2":"I am a string", "new key":"New value", "key1":[1,2,3],123:456}

**Functions**

Python functions can be used to abstract pieces of code to use elsewhere.

Syntax

def function\_name(parameters):

# Some code here

Example

def add\_two(a, b):

c = a + b

return c

# or without the interim assignment to c

def add\_two(a, b):

return a + b

Syntax

def function\_name(parameters, named\_default\_parameter=value):

# Some code here

Example

def shout(exclamation="Hey!"):

print exclamation

shout() # Displays "Hey!"

shout("Watch Out!") # Displays "Watch Out!"

**Function Objects**

Python functions are first-class objects, which means that they can be stored in variables and lists and can even be returned by other functions.

Example

# Storing function objects in variables:

def say\_hello(name):

return "Hello, " + name

foo = say\_hello("Alice")

# Now the value of 'foo' is "Hello, Alice"

fun = say\_hello

# Now the value of 'fun' is a function object we can use like the original function:

bar = fun("Bob")

# Now the value of 'bar' is "Hello, Bob"

Example

# Returning functions from functions

# A simple function

def say\_hello(greeter, greeted):

return "Hello, " + greeted + ", I'm " + greeter + "."

# We can use it like this:

print say\_hello("Alice", "Bob") # Displays "Hello, Bob, I'm Alice."

# We can also use it in a function:

def produce\_greeting\_from\_alice(greeted):

return say\_hello("Alice", greeted)

print produce\_greeting\_from\_alice("Bob") # Displays "Hello, Bob, I'm Alice."

# We can also return a function from a function by nesting them:

def produce\_greeting\_from(greeter):

def greet(greeted):

return say\_hello(greeter, greeted)

return greet

# Here we create a greeting function for Eve:

produce\_greeting\_from\_eve = produce\_greeting\_from("Eve")

# 'produce\_greeting\_from\_eve' is now a function:

print produce\_greeting\_from\_eve("Alice") # Displays "Hello, Alice, I'm Eve."

# You can also invoke the function directly if you want:

print produce\_greeting\_from("Bob")("Eve") # Displays "Hello, Eve, I'm Bob."

Example

# Using functions in a dictionary instead of long if statements:

# Let's say we have a variable called 'current\_action' and we want stuff to happen based on its value:

if current\_action == 'PAUSE':

pause()

elif current\_action == 'RESTART':

restart()

elif current\_action == 'RESUME':

resume()

# This can get long and complicated if there are many values.

# Instead, we can use a dictionary:

response\_dict = {

'PAUSE': pause,

'RESTART': restart,

'RESUME': resume

}

response\_dict[current\_action]() # Gets the correct function from response\_dict and calls it

**len()**

Using len(some\_object) returns the number of \_top-level\_ items contained in the object being queried.

Syntax

len(iterable)

Example

>> my\_list = [0,4,5,2,3,4,5]

>> len(my\_list)

7

>> my\_string = 'abcdef'

>> len(my\_string)

6

**List Comprehensions**

Convenient ways to generate or extract information from lists.

Syntax

[variable for variable in iterable condition]

[variable for variable in iterable]

Example

>> x\_list = [1,2,3,4,5,6,7]

>> even\_list = [num for num in x\_list if (num % 2 == 0)]

>> even\_list

[2,4,6]

>> m\_list = ['AB', 'AC', 'DA', 'FG', 'LB']

>> A\_list = [duo for duo in m\_list if ('A' in duo)]

>> A\_list

['AB', 'AC', 'DA']

**Lists**

A Python data type that holds an ordered collection of values, which can be of any type. Lists are Python's ordered mutable data type. Unlike tuples, lists can be modified in-place.

Example

>> x = [1, 2, 3, 4]

>> y = ['spam', 'eggs']

>> x

[1, 2, 3, 4]

>> y

['spam','eggs']

>> y.append('mash')

>> y

['spam', 'eggs', 'mash']

>> y += ['beans']

>> y

['spam', 'eggs', 'mash', 'beans']

**Loops**

**For Loops**

Python provides a clean iteration syntax. Note the colon and indentation.

Example

>> for i in range(0, 3):

>> print(i\*2)

0

2

4

>> m\_list = ["Sir", "Lancelot", "Coconuts"]

>> for item in m\_list:

>> print(item)

Sir

Lancelot

Coconuts

>> w\_string = "Swift"

>> for letter in w\_string:

>> print(letter)

S

w

i

f

t

**While Loops**

A While loop permits code to execute repeatedly until a certain condition is met. This is useful if the number of iterations required to complete a task is unknown prior to flow entering the loop.

Syntax

while condition:

//do something

Example

>> looping\_needed = True

>>

>> while looping\_needed:

>> # some operation on data

>> if condition:

>> looping\_needed = False

**print**

A function to display the output of a program. Using the parenthesized version is arguably more consistent.

Example

>> # this will work in all modern versions of Python

>> print("some text here")

"some text here"

>> # but this only works in Python versions lower than 3.x

>> print "some text here too"

"some text here too"

**range()**

The range() function returns a list of integers, the sequence of which is defined by the arguments passed to it.

Syntax

argument variations:

range(terminal)

range(start, terminal)

range(start, terminal, step\_size)

Example

>> range(4)

[0, 1, 2, 3]

>> range(2, 8)

[2, 3, 4, 5, 6, 7]

>> range(2, 13, 3)

[2, 5, 8, 11]

**Sets**

Sets are collections of unique but unordered items. It is possible to convert certain iterables to a set.

Example

>> new\_set = {1, 2, 3, 4, 4, 4,'A', 'B', 'B', 'C'}

>> new\_set

{'A', 1, 'C', 3, 4, 2, 'B'}

>> dup\_list = [1,1,2,2,2,3,4,55,5,5,6,7,8,8]

>> set\_from\_list = set(dup\_list)

>> set\_from\_list

{1, 2, 3, 4, 5, 6, 7, 8, 55}

**Slice**

A Pythonic way of extracting "slices" of a list using a special bracket notation that specifies the start and end of the section of the list you wish to extract. Leaving the beginning value blank indicates you wish to start at the beginning of the list, leaving the ending value blank indicates you wish to go to the end of the list. Using a negative value references the end of the list (so that in a list of 4 elements, -1 means the 4th element). Slicing always yields another list, even when extracting a single value.

Example

>> # Specifying a beginning and end:

>> x = [1, 2, 3, 4]

>> x[2:3]

[3]

>> # Specifying start at the beginning and end at the second element

>> x[:2]

[1, 2]

>> # Specifying start at the next to last element and go to the end

>> x[-2:]

[3, 4]

>> # Specifying start at the beginning and go to the next to last element

>> x[:-1]

[1, 2, 3]

>> # Specifying a step argument returns every n-th item

>> y = [1, 2, 3, 4, 5, 6, 7, 8]

>> y[::2]

[1, 3, 5, 7]

>> # Return a reversed version of the list ( or string )

>> x[::-1]

[4, 3, 2, 1]

>> # String reverse

>> my\_string = "Aloha"

>> my\_string[::-1]

"aholA"

**str()**

Using the str() function allows you to represent the content of a variable as a string, provided that the data type of the variable provides a neat way to do so. str() does not change the variable in place, it returns a 'stringified' version of it. On a more technical note, `str()` calls the special `\_\_str\_\_` method of the object passed to it.

Syntax

str(object)

Example

>> # such features can be useful for concatenating strings

>> my\_var = 123

>> my\_var

123

>> str(my\_var)

'123'

>> my\_booking = "DB Airlines Flight " + str(my\_var)

>> my\_booking

'DB Airlines Flight 123'

**Strings**

Strings store characters and have many built-in convenience methods that let you modify their content. Strings are immutable, meaning they cannot be changed in place.

Example

>> my\_string1 = "this is a valid string"

>> my\_string2 = 'this is also a valid string'

>> my\_string3 = 'this is' + ' ' + 'also' + ' ' + 'a string'

>> my\_string3

"this is also a string"

**Tuples**

A Python data type that holds an ordered collection of values, which can be of any type. Python tuples are "immutable," meaning that they cannot be changed once created.

Example

>> x = (1, 2, 3, 4)

>> y = ('spam', 'eggs')

>> my\_list = [1,2,3,4]

>> my\_tuple = tuple(my\_list)

>> my\_tuple

(1, 2, 3, 4)

**Tuple Assignment**

Tuples can be expanded into variables easily.

Example

name, age = ("Alice", 19)

# Now name has the value "Alice" and age has the value 19

# You can also omit the parentheses:

name, age = "Alice", 19

**Variables**

Variables are assigned values using the '=' operator, which is not to be confused with the '==' sign used for testing equality. A variable can hold almost any type of value such as lists, dictionaries, functions.

Example

>> x = 12

>> x

12