**Joshua Barnard – CS 232\_Spring ‘16**

Python is Procedural and Functional, Strong (type of object does not change, even if value is mutable), and Interpreted.

**Differentiate Base**:0b\_ 🡨 binary (2)

0o\_ 🡨 octal (8)

0h\_ 🡨 hex (16)

**Data Abstraction** 🡨 The ability to manipulate data w/out knowledge of the data’s internal structure/workings.

**Variables** 🡨 Names which refer to values in the computers memory that you can define for use with your program. Can only contain Upper & Lowercase letters,digits,&underscores.

**Names** 🡨 Refers to a thing, rather than the thing itself. Assignment does not copy a value, it just attaches a name. Cannot begin with a digit.

**Commands:**

**\ (Escape)** 🡨 Causes the following character to be interpreted as an effect.

**del \_\_** 🡨Removes local variables, or elements from local variables.

**type (\_)** 🡨 Returns the inputted object’s type.

**def** 🡨 Function Definition

**class**  🡨 Class definition

**len(\_)** 🡨 Built-in function. Counts and returns the number of characters in a string, items in list, etc.

**sorted()** 🡨 Displays the sorted list

**print() 🡨** Prints to screen for user. Has no effect on the execution of the program, does not assign a value to a variable, nor does it return a value from a function call. For Humans.

**return** 🡨 Used in the ongoing execution of the program. Can involve assigning a value to a variable, appending lists, or adding to dictionaries. For the Program.

**Atomic Data Types:**

**int**: Integer #. (Add .5 to round up.)

**float:** Floating Point #

**str:** Sequence of text characters.

**bool:** Boolean expressions evaluate one of two states as true or false. False == None, False, 0, Empty Collections.

**Molecular Data Types:**

**Lists [ ]:** Mutable; Ordered; Item values can be changed.

**Tuples ( ):** Ordered, not mutable.

**Sets { }:** Unordered; UnIndexed.

**Dictionaries{\_:\_}:** Unordered; Indexed

**Lists:**

**list()** 🡨 turns object into a list.

**\*.sort()** 🡨 Sorts and resaves the list.

**\*.sorted() 🡨** Will display the sorted list without saving.

**\*.count(\_)** 🡨 Counts # of \_ w/in a list.

**\*.extend(\_)**🡨 Adds the \_ list to the \* pre-existing list.

**\*.insert(#,”\_”)**🡨Add \_ to # spot in list.

**\*.append(“\_”)** 🡨 Adds \_ to list.

**\*.remove(\_)** 🡨 Removes \_ from list.

**Tuples:**

**tuple() 🡨** Turns object into tuple.

**cmp(tuple1, tuple2)** 🡨 Compares elements of both tuples.

**max()** 🡨 returns integer with max value from tuple

**min() 🡨** returns integer with minimum value from tuple.

**Dictionaries:**

**len(dict)** 🡨 Total Length of dictionary

**str(dict)** 🡨 Produces printable string representation of a dictionary

**get() 🡨** Works unless the key doesn’t exist, then it returns empty.

**defaultdict()** 🡨 Searches for key not in dict, when the dictionary is created. Creates an entry with key, and default value. From collections module.

**\*.update(Index : “newValue”)** 🡨 Adds value to the \* dictionary, or replaces a value in the \* dictionary.

**\*.clear()** 🡨 Removes all elements.

**\*.items()** 🡨 Returns list of dict’s tuple pairs (key, value).

**\*.keys()** 🡨 Returns list of dict key’s

**\*.values(dict)** 🡨 Returns list of dict’s values.

**Sets:**

**\*.add(\_)** 🡨 adds \_ item(s)

**\*.remove(\_)** 🡨 remove \_ item(s)

**Operations (Molecular):**

del Dictionary[‘Index’]

Dictionary[‘Index’] = “newValue”

set() 🡨 creates a null set

**Operators:**

**+** 🡨 Addition

**-**  🡨 Subtraction

**\*** 🡨 Multiplication

**\*\*** 🡨 Exponentiation

**/**  🡨 Floating pt division

**//**  🡨 Integer division

**%** 🡨 Mod

**Logical Operators:**

**& (AND**) 🡨 Intersection

**|**  🡨 Union

**^ (OR)** 🡨 XOR (in one but not both)

**-**  🡨 Difference

**NOT()** 🡨 Not

**Comparison Operators:**

**==** 🡨 Comparison

**!= (<>)** 🡨 Does Not Equal

**and**  🡨 AND

**or**  🡨 OR

**>**  🡨 Less Than

**>=**  🡨 Less than or Equal to

**<**  🡨 More Than

**<=**  🡨 More Than or Equal to

**Assignment Operators:**

**=** 🡨 Assignment.

**+=** 🡨 Add then save result.

**-=** 🡨 Subtract then save result.

**\*=** 🡨 Multiply then save result.

/= 🡨 Divide then save result.

%= 🡨 Mod then save result.

\*\*= 🡨 Exponentiate then save result.

//= 🡨 Int division then save result.

**Functions:**

**Type Casting** 🡨 Temporarily converts one data type to another; ie: (float)one/(float)two == float(one)/float(two)

**\*args** 🡨 Allows you to pass any # of arguments to a function. Used to send a non-keyworded variable length argument list to the function.

**\*\*kwargs** 🡨 Allows you to pass keyworded of any length to a function. Used to handle named arguments in the function

**Objects:**

**Objects are**: Data Definition, &

Structure Definition.

**Modules are** objects.

Everything is an object, including: bools, ints, floats, strings, data structures, functions and programs.

**Class** 🡨 The Definition of an object.

An instance contains value(s).

Ex: Object: Name Record.

Instance : “Barnard” “Joshua”

**Constructor** 🡨 Automatically invoked when an object comes into existence. The same name as the class, except w/out a return type (not even void).

**Destructor** 🡨 Executes when an object goes out of existence. They put tilde’s (~) infront of objects; ie: ~ClassName.

**Mutable** 🡨data values can be changed.

**Immutable** 🡨 data values are constant.

**Looping:**

**while** 🡨 Compares two values and continues if count is less than or equal to final count value.

**for** 🡨 Compares two values and continues until condition is met. *Ex:* For x in y.

**else** 🡨 If break is not called, the else statement will run.

**continue** 🡨 go straight to the next iteration of the loop.

**break**  🡨 exits the loop

**Conditionals:**

**if**  🡨 Checks whether or not a condition is True

**elif**  🡨 Checks whether or not a condition is True after a False if statement.

**Functions:**

**Functions** 🡨 Named piece of code, separate from all others, to be reused. They take in input parameters and returns any number or type.

Functions can be passed to functions.

Functions can be returned by functions.

**Define** 🡨 def func\_name():

**Call** 🡨 func\_name()

**Arguments**

Two ways to assign argument values to function parameters:

**Positional** 🡨 Do not have keywords and are assigned first. Value are copied to their corresponding parameter in order.

Ex: menu('Syrah', 'Pork', 'Cake')

**Keyword** 🡨 Have keywords and are assigned second, after positional. To avoid positional confusion, values are added by specifying the name of corresponding parameter.

Ex: menu(entree='beef', dessert='pie', wine='zinfandel')

\* 🡨 Groups a variable number of positional arguments into a tuple of parameter values.

**\*args** 🡨 Allows you to pass any # of arguments to a function. Used to send a non-keyworded variable length argument list to the function.

\*\* 🡨Groups keyword arguments into a dictionary, where the argument names are the keys, and their values are the corresponding dictionary values.

**\*\*kwargs** 🡨 Allows you to pass keyworded of any length to a function. Used to handle named arguments in the function.

**Comprehensions:**

**Comprehensions** 🡨 A compact way of creating a data structure from one or more iterators. They make it possible to combine loops and conditional tests with less verbose syntax. They can quickly build list-type structures in an algorithmic way, by embedding code that can build and conditionally select.

A compact way of creating a data structure from one or more iterators. They make it possible to combine loops and conditional tests with less verbose syntax.

Ex: [number for number in range (#, #)]

[ *expression* for *item* in *iterable* ]

{*key* : *value* for *expression* in *iterable*}

{*expression* for *expression* in *iterable*}

**Iterators:**

for() 🡨 Similar to a while loop.

Ex: word = 'cat' **for** letter **in** word: **print**(letter)

range(start, stop, step) 🡨 Returns a stream of numbers w/in a specified range, w/out needing to create and store a large data structure such as a list or tuple. Returns an iterable object.

Ex: **for** x **in** range(0,3): print(x)

zip() 🡨 Stops iterating when then shortest sequence is done. Returns an iterable object.

Ex: **for** day, fruit, drink, dessert **in** zip(days, fruits, drinks, desserts): **print**(day, ": drink", drink, "- eat", fruit, "- enjoy", dessert)

**Python Reserved Words:**

False class finally nonlocal if

True raise for lambda in

None def from return is

while and del global as

with not elif import or

yield assert else except

pass break try continue

**Counters**

**Counter()** 🡨 A dictionary subclass container (generator type object) which contains the number of times a certain hashable object (element) exists within a list or tuple. From collections module.

**Local and Global Scope**

**Local variables** are defined within a function and cannot be accessed outside of said function.

**Global variables** are defined outside of any particular function. They must be assigned using the command global.

**Name Collision** 🡨 Local scope supersedes global scope.

**global \_\_\_** 🡨 Uses the global scope variable.

**locals()** 🡨 Everything at the local scope

**globals()** 🡨 Everything at the global scope.

**id() 🡨** Returns the unique id# of an identifier (object).

**File Input / Output**

**once()** 🡨 Once a dictionary is set, the once() function creates a file object that is connected to the file on disk.

It takes in two arguments: the name of the file and a 2-letter string on how to open it.

**First letter is**: ‘r’ for read

‘w’ for write (overwrite)

‘a’ for write (append)

‘x’ for write (create new file)

**Second Letter is**: ‘t’ for text

‘b’ for binary

**Methods:** \_\_\_.**read()** 🡨 Extracts the text as a string (continuing from last read action) of a file.

\_\_\_.**close()** 🡨 Closes the file preventing it from being read or modified.

\_\_\_.**readline()** 🡨Extracts the text as a string (continuing from last read action) until the end of the line.

\_\_\_.**readlines()**🡨 Extracts the entirety of the text as a list of lines.

**\_\_\_.tell()** 🡨 Displays your current position within the file.

**\_\_\_.seek()** 🡨 Changes your position within the file.

**Objects**

**Objects** **are**: Data Definition, &

Structure Definition.

Modules are objects.

Everything is an object, including: bools, ints, floats, strings, data structures, functions and programs.

**Class** 🡨 The Definition of an object.

An **instance** contains value(s).

Ex: Object: Name Record.

Instance : “Barnard” “Joshua”

**Constructor** 🡨 Automatically invoked when an object comes into existence. The same name as the class, except w/out a return type (not even void).

**Destructor** 🡨 Executes when an object goes out of existence. They put tilde’s (~) infront of objects; ie: ~ClassName.

**Mutable** 🡨data values can be changed.

**Immutable** 🡨 data values are constant.

**Functions**

**Functions** 🡨 Named piece of code, separate from all others, to be reused. They take in input parameters and returns any number or type.

Functions can be passed to functions.

Functions can be returned by functions.

**Define** 🡨 def func\_name():

**Call**  🡨 func\_name()

**Type Casting** 🡨 Temporarily converts one data type to another; ie: (float)one/(float)two == float(one)/float(two)

**Inner Functions** 🡨 Functions as objects, ie functions within functions.

**Decorator Functions** 🡨 Designed to take in a function as input and pass back a function which “decorates” it, by showing debugging info or “wrapping” it.

**\_\_init\_\_ 🡨** Built-in Method, a constructor to create an object.

**Iterators**

**zip()** 🡨 Creates a tuple containing one element from each list in order until one of the lists has been exhausted.

**Python3 Formatting**

**\_\_.format(\_)** 🡪 Python3 uses {} as a placeholder in which to insert the formatting information. Then you use

Pattern Matching can be done with the re module and method: re.match()

**match()** 🡨 will generate a “match object” if the given text matches the beginning of the given string. No object is given if no text matches.

**group()** 🡨 method of match object. Extracts data from match object.

**findall()** 🡨 finds all matches.

**search()** 🡨 finds only the first one.

**sub()** 🡨 mutates the string being searched.

**\_.\*** 🡨 finds all strings starting with \_

**\_.\*-** 🡨 finds all strings starting with \_ and ending with -.

**\_.\*?-** 🡨 Specifies to match the shortest possible string, instead of the default of the longest.

**{n}** 🡨 number of repetitions. ie: P{2} == PP.

**.\*?tt.\*?** 🡨 entire words containing tt.

**[a-z]** 🡨 match a single lower-case letter between a and z.

**{0:7.2f} 🡨** 7 characters, with 2 decimal points, floating point.

'{} {}'.format(1, 2)

{} means just the next positional argument, with default format;

{0} means the argument with index 0, with default format;

{:d} is the next positional argument, with decimal integer format;

{0:d} is the argument with index 0, with decimal integer format.

d = decimal, f = floating point

**Sample Code**

**File I/O**

import os

os.getcwd() 🡪’C:\\’

os.chdir(“C:\\”)

myfile = open("a\_file.txt", "xt")

myfile.tell() 🡪 0

for i in range(0,9):

LineOfText = myfile.readline()

print("Line # ",i,":",LineOfText,end='')

myfile.write(“Woo Hoo!”)

myfile.seek(1)

myfile.read(7) 🡪’oo Hoo!’

myfile.close()

import sys

for i in sys.path:

print(i)

sys.path.append(“U:\\”)

**Exception Handling**

try:

myfile = open("a\_file.txt", "xt")

except:

print("Whoa, fella, that file is already here!")

try:

a = b + 3

except:

print("b doesn’t exist”)

**Classes**

Class Die():

Def \_\_init\_\_ (self, num\_sides):

self.num\_sides = num.sides

self.value = 0

class Dwarf():

def \_\_init\_\_(self, name, fname, height, axe):

self.name = name

self.fname = fname

self.height = height

axe = axe.lower()

if self.axe == True:

self.axe = axe

def introduce(self):

print(‘I am{}, son of {}!’.format(self.name, self.fname)

print(‘I am’, self.height, ‘ inches tall, and it is ‘, self.axe, ‘ that I own an axe!”)

def double(x):

return 2\*x

class Person(Person):

def \_\_init\_\_(self, name, id\_num, major):

super().\_\_init\_\_(name)

self.id\_num = id\_num

self.major = major

def change\_major(self, new\_major):

self.major = new\_major

def introduce(self):

print("Hi, I'm " + self.name + "and my major is " + self.major

**Importing**

import \_package\_

from \_package\_ import \_module(s)\_

**Scope of Variables**

x = 7

def a\_func(a\_val):

x = 5 #local x variable = 5

global x = 5 #changes global x var.

**Comprehensions:**

a\_list = [number **for** number **in** range(1,6) **if** number % 2 == 1]

a\_list

[1, 3, 5]

rows = range(1,4)

cols = range(1,3)

cells = [(row, col) **for** row **in** rows **for** col **in** cols]

**for** cell **in** cells: **print**(cell)

🡪(1, 1) (1, 2) (2, 1) (2, 2) (3, 1) (3, 2)

[number for number in range (0,10)]

🡪[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

{number: number + 3 for number in range (0,10)}

🡪{0: 3, 1: 4, 2: 5, 3: 6, 4: 7, 5: 8, 6: 9, 7: 10, 8: 11, 9: 12}

{number: number + 3 for number in range (0,10) if number % 2 == 1}

🡪{1: 4, 3: 6, 9: 12, 5: 8, 7: 10}

**Counters**

import collections

file = "Hello World"

collections.Counter(file)

🡪Counter({'l': 3, 'o': 2, 'r': 1, 'e': 1, 'H': 1, 'd': 1, ' ': 1, 'W': 1})

breakfast = ['spam', 'spam', 'eggs', 'spam']

breakfast\_counter = Counter(breakfast)

>>> breakfast\_counter

Counter({'spam': 3, 'eggs': 1})

**Formatting**

'{:06.2f}'.format(3.141592653589793)

>>> 3.14

My\_name = “Fred”

‘My name is {}’.format(my\_name)

🡪 ‘My name is Fred’

‘. . . {} . . . {} . . . ‘.format(p=’a’,q=’b’)

🡪 ‘. . . a . . . b . . . ‘

a\_list = ["Leonardo", "Kate"]

'The actor was {0[1]}'.format(a\_list, character, year)

🡪 'The actor was Kate'

a\_dict = {'actor': 'Leonardo', 'actress': 'Kate'}

'In Titanic, {0[actor]} kissed {0[actress]}'.format(a\_dict)

'In Titanic, Leonardo kissed Kate'

'In the year {y}, {p} played {c}'.format(p=person, c=character, y=year)

🡪 'In the year 1912, Leonardo played Jack'

import re

infile = open("a\_file.txt", 'rt')

ga = infile.read()

match\_object = re.match('F.\*', ga)

if match\_object:

print(match\_object.group())

ga = re.sub('a','!', ga)

x = 17.18665

print(“x equals {:.2f}.”.format(x))

'{1} {0}'.format('one', 'two')

>>> two one

**Constructor**

class Person():

def \_\_init\_\_(self, name):

self.name = name

**Decorator Functions**

def get\_text(name):

return "Output of the name " + name

def p\_decorate(func):

def func\_wrapper(name):

return "<p>"+func(name)+"</p>"

return func\_wrapper

type(p\_decorate(get\_text))

🡪 <class 'function'>

my\_get\_text = p\_decorate(get\_text)

my\_get\_text("john")

🡪'<p> Output of the name john </p>'

def document\_it(func):

def new\_function(\*args, \*\*kwargs):

print('Running the function:', func.\_\_name\_\_)

print('Positional arguments:', args)

print('Keyword arguments:', kwargs)

result = func(\*args, \*\*kwargs)

print('Result:', result)

return result

return new\_function

def sum\_of\_squares(list\_of\_nums):

return sum(a \* a for a in list\_of\_nums)

annotated\_sum\_of\_squares = document\_it(sum\_of\_squares)

annotated\_sum\_of\_squares((1, 2, 3, 4))

🡪Running the function: sum\_of\_squares

🡪Positional arguments: ((1, 2, 3, 4),)

🡪Keyword arguments: {}

🡪Result: 30

🡪30

@document\_it

def sum\_of\_cubes(list\_of\_nums):

return sum(a \* a \* a for a in list\_of\_nums)

sum\_of\_cubes((1, 2, 3, 4))

🡪Running the function: sum\_of\_cubes

🡪Positional arguments: ((1, 2, 3, 4),)

🡪Keyword arguments: {}

🡪Result: 100

🡪100

def run\_and\_print\_result(func):

def new\_func(\*args, \*\*kwargs):

result = func(\*args, \*\*kwargs)

print(‘I ran the function ‘, func.\_\_name\_\_)

print(‘Return value is ‘, result)

return new\_func

@return\_and\_print\_result

def decorate\_it(func):

def inner(\*args, \*\*kwargs)

result = func(\*args, \*\*kwargs)

print(“result is {}.format(result))

return result

return inner

@classmethod

def num\_students(cls):

print("There are ", cls.count, " student entered.")

**Extra Code**

from collections import defaultdict

periodic\_table = defaultdict(int)

periodic\_table['H'] = 1

periodic\_table['He'] = 2

for index, data in quick\_dict.items():

if index == "C": print("Found data item ", data)