Shawn Fenner

INF 473 – Problems in Informatics

Progress Report

July 25, 2018

Prepared for

David Gray

Fort Hays State University

Summer Semester 2018

Preface

Throughout the summer semester, I have worked with the Python programming language in order to learn its many features and become more proficient in its use. Python is a popular language, due in part to its simple syntax, implicit memory management, expansive list of modules, and portability. These characteristics promote a quicker, easier-to-understand development cycle and make such a language suitable for a variety of applications, from scientific computing to video games and web development as well as prototyping.

Python Proficiency

This section details the features of the Python programming language that I have learned and/or expanded upon during my research and coding this semester, along with a short example of each (in red and blue text) demonstrating my understanding:

* The **unpack operators** (\* and \*\*) support sending a variable number of arguments or keyword arguments, respectively, to a function or method.

# print variable length arguments

def unpackVals(formalArg, \*args, \*\*kwargs):

print(formalArg) # a.k.a positional arg

if len(args):

print("variable length arguments:")

for a in args:

print(" ", a)

if len(kwargs):

print("variable length keyword arguments:")

for key, value in kwargs.items():

print(" key: {} - value: {}".format(key, value))

unpackVals([99, 125], \*(44, b'\xff', float.fromhex('0x1.a7p4'),),

name='Shawn', color='0x8000ff', hexFloat = float.hex(26.4375),

\*\*{ "intVal": 9808218, 'school': 'FHSU'})

* **Function decorators**, prefixed with the @ symbol, allow modification of a function’s behavior.
  + From the functools module, the **wraps function decorator** can be imported which ensures a decorated function’s name and docstring are not overridden.

# use a decorator to print the time required to run a function

import time

from functools import wraps

def timeDecorator(fn):

@wraps(fn)

def wrapper():

time1 = time.time()

fn()

time2 = time.time()

return "Time to run function - {}: {}\n".format(fn.\_\_name\_\_,

time2 - time1)

return wrapper

@timeDecorator

def addList1():

numList = []

for num in range(100000):

numList.append(num)

print("{} - sum: {}".format(addList1.\_\_name\_\_,

sum(numList)))

@timeDecorator

def addList2():

print("{} - sum: {}".format(addList2.\_\_name\_\_,

sum([num for num in range(100000)])))

@timeDecorator

def addList3():

print("{} - sum: {}".format(addList3.\_\_name\_\_,

sum((num for num in range(100000)))))

print(addList1()); print(addList2()); print(addList3())

* + The **@staticmethod** decorator does not implicitly pass the self, cls, or any other arguments and creates a method inside a class definition that can be called without instantiating a class object.

# sample usage of the static method decorator

# used in a class in Canvas Knight

class Cloud(object):

\_clouds = list()

@staticmethod

def updateAll(dt):

if len(Cloud.\_clouds):

for c in Cloud.\_clouds:

c.update(dt)

if Cloud.\_clouds[0].done:

Cloud.\_clouds.pop(0)

else:

return

...

* + The **@classmethod** decorator implicitly receives the type of the class as its argument. This feature is useful in creating factory methods and creating multiple constructors.

# This class defines a class method that is used as an alternate

# constructor when the age of the video game is not known

from datetime import date

class VideoGame(object):

def \_\_init\_\_(self, title, genre, age):

self.title = title; self.genre = genre; self.age = age

self.isClassic = VideoGame.isClassic(self.age)

@classmethod

def fromReleaseDate(cls, title, genre, releaseDate):

return cls(title, genre, date.today().year - releaseDate)

def \_\_str\_\_(self):

return "title: {} - genre: {} - age: {} - isClassic: {}".format(self.title, self.genre, self.age, self.isClassic)

@staticmethod

def isClassic(age):

return age >= 20

game1 = VideoGame("Super Mario World", "Action", 27)

game2 = VideoGame.fromReleaseDate("Mega Man 2", "Action", 1988)

print(game1, game2, sep='\n')

* **List comprehensions** provide an efficient and concise way to create lists.

# create a multidimensional list using a list comprehension

list([(x // 2, y) for x in range(0, 101, 20)] for y in range(3))

* **Generators**, while similar to list comprehensions, do not create an entire list of objects in memory, rather only creating one object on demand at a time, resulting in less memory consumption. Generators produce the same results as a generator function containing a yield statement and returning a generator object, although the latter might be preferred when writing complex statements

# create a generator object then iterate over it,

# printing each value in turn

def yieldStuff(\*args):

for a in args:

yield a \* a

for r in yieldStuff(2, 5, 11)

print(r)

* Understanding the **general container classes** (lists, tuples, dictionaries, and sets) and their differences is important to using Python effectively and deciding which one to rely on for a given task.

# list – mutable and allocates additional memory for future append calls

testList = [5, -9, 44.2, 'Shawn', None]

# tuple – immutable, fixed-size, and thus more memory efficient than a list

testTuple = ('Python', 0, testList[-1])

# dictionary – keys correspond to values

testDict = { 'School': 'FHSU', 49: True, 'ints': (16, 77, 85,) }

# set – can’t contain duplicate values

testSet = { 16, 0.49801, 'Shawn', 16 }

* **Specialized containers** (namedtuple; deque; Counter; orderedDict; defaultdict; ChainMap; etc.) build upon and provide additional functionality to the general containers without the programmer having to take the time to implement similar features.
  + **Named tuples** are similar to the standard tuple container but have field values accessible by name lookup as well as standard indexing.

# specialized container type - namedtuple

from collections import namedtuple

# create the namedtuple object

EmpRecord = namedtuple('EmployeeRecord',

'name, age, title')

# instantiate a list of employee records

employees = [EmpRecord('Fred', 65, 'Software Developer'),

EmpRecord('Tim', 32, 'Level Designer'),

EmpRecord('Thomas', 44, 'Web Developer')]

# unpack a record and assign to variables just like a

# standard tuple

name, age, title = employees[0]

# fields are accessible by index, name, or the getattr function

print(employees[0][0])

print(employees[0].name)

print(getattr(employees[0], 'name'))

* + **Deques** support rapid appends to either side of the container and allow for a fixed container size to be defined upon instantiation. However, lists will be faster for random access.

# specialized container type - deque

from collections import deque

from copy import deepcopy

d = deque((5, 7, -11), maxlen=5)

d.appendleft(‘somStr’)

d.append([1.199999, 2.444178])

# deque is now full and appending a new element will

# remove the right most element from the deque

d.append(Rect(0, 0))

# create a shallow copy

dCopy = d.copy()

# create a deep copy

dCopy = deepcopy(d)

# remove the rightmost element

dCopy.pop()

# rotate all elements in the deque to the left

d.rotate(-4)

# extend right n elements and remove that many from

# the left side if the deque is full

d.append([99, 100])

# attempt to insert an element into a specified position

# raises an IndexError exception if the deque is full

d.insert(2, object())

* + **Counters** keep a count for each hashable item (key/value pair) they contain.

# create a new Counter from an iterable

numCounter = Counter('1231451')

print(numCounter)

# create a new Counter from kwargs

vehicleCounter = Counter(boats = 4, trucks = 5, planes = 2, trains = 3)

# remove an element from a Counter by deleting it

del vehicleCounter['boats']

# returns 0 if key is not in Counter

print(vehicleCounter['boats'])

* The **reversed function** allows backwards iteration of a container. The slice function can perform the same functionality, although reversed is slightly faster.

# iterate list in reverse order based on ternary operator

myList = [1, 5, 99]; p1Y = 0; p2Y = 50

print(\*(myList if p1Y > p2Y else reversed(myList)))

print(\*(myList if p1Y > p2Y else myList[::-1]))

* **Shallow copies** allow one container to contain the exact same items as another. This concept is also known as aliasing and changing the contents of one container will change that of the other.

listA = list()

listB = [55, 44, 22]

# create a shallow copy; listA now points to listB; altering any

# of listA's elements (and vice versa) will affect listB's elements

listA = listB

print("listA:", listA, "listB:", listB)

listB[0] = 999

print("listA:", listA, "listB:", listB)

# in a shallow copy, both lists will have their first element deleted

del listB[0]

print("listA:", listA, "listB:", listB)

# in a shallow copy, both lists will receive the same Rect object

listA.insert(0, Rect(1, 1))

print("listA:", listA, "listB:", listB)

# in a shallow copy, modifying an object will affect both lists

listA[0].x = 92; listA[0].y = 144;

print("listA:", [str(val) for val in listA],

"listB:", [str(val) for val in listB])

* **Deep copies** allow one container to contain copies of the items of another. Unlike shallow copying, changing the contents of either container will not affect those in the other.

listA = list()

listB = [55, 44, 22]

# create a deep copy; listA won't point to listB and altering its

# elements won't affect the elements in listB

listA = listB[:] # splice; faster than the line below

listA = listB.copy() # same result as the line above

* Python’s **magic methods** can be overridden to define custom functionality. Several of the many magic methods available include those for class construction and destruction (e.g. \_\_new\_\_; \_\_del\_\_; \_\_init\_\_), comparison (e.g. \_\_cmp\_\_; \_\_ne\_\_; \_\_ge\_\_), mathematical functions (e.g. \_\_abs\_\_; \_\_round\_\_; \_\_ceil\_\_), bit manipulation (e.g. \_\_lshift\_\_; \_\_and\_\_; \_\_xor\_\_), type conversion (e.g. \_\_float\_\_; \_\_hex\_\_; \_\_trunc\_\_) and finally class representation (e.g. \_\_str\_\_; \_\_dir\_\_; \_\_format\_\_).

class Enemy(Rect):

...

# define a customized return string for this class when an

# instance of it is type-cast to a string

def \_\_str\_\_(self):

return "name: {} hp: {} ".format(self.name, self.hp) + super().\_\_str\_\_()

...

* The **debugging module, pdb**, aids the programmer in examining the call stack, setting breakpoints, and single-stepping through code. Several commands available include: s[tep] for executing the current line of code, c[ontinue] for continuing program execution until the next breakpoint is encountered, and w[here] for printing a stack trace.

import pdb

def stackViewA():

# set\_trace hard-codes a break point at which point the programmer

# can enter various pdb commands in order to debug the program

pdb.set\_trace()

def stackViewB():

stackViewA()

def stackViewC():

stackViewB()

stackViewC()

* The **dir function** attempts to retrieve the attributes of an object and place the names of them into a list. User-defined objects can override the \_\_dir\_\_ magic method for custom functionality.

class Person(object):

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

self.age = age; self.name = name; self.state = state

# override the \_\_dir\_\_ magic method

def \_\_dir\_\_(self):

return ['age', 'name', 'state']

student = Person('Toby', 26, 'WA')

print(dir(student))

* **Lambdas** are single expression functions that always return a value; similar to anonymous functions in JavaScript (notable though is that all functions in Python return a value of None even when there is no return statement). Lambdas are useful when working with functions such as map, filter, and reduce.

# return the sum of two arguments

sumArgs = lambda arg1, arg2: arg1 + arg2

* The **map function** calls a function for each item in a container.

# multiply and power each element in the list to itself

for i in range(0, 6):

values = list(map(lambda fn: fn(i),

(lambda val: val \* val, lambda val: val \*\* val)))

print(i, values, sep=': ')

* The **filter function** creates a filter object that contains the contents of a container whose objects evaluated to True during the filter’s conditional function calls.

# extend a list with odd numbers from a tuple

a = [1, 10, 100]; b = [29, 20, 211, 65, 64]

a.extend(filter(lambda val: val % 2, b))

* The **reduce function** returns a result based on the combined computations of each item in a container.

from functools import reduce

def addNums(x, y):

print("adding {} and {}".format(x, y))

return x + y

total = reduce(addNums, range(1, 6))

print("total: %d" % total) # total is now 15

* The **zip function** allows combining of two or more containers into a single zip object, allowing concurrent iteration of multiple containers.

listA = [87891, 55291, 99402]

listB = ['Shawn', 'Fred', 'Timmy']

# create a new list using a zip iterable with the contents of two lists

records = list(zip(listA, listB))

print(records)

# zip also allows unpacking of elements into separate tuples

ids, names = zip(\*records)

print(ids, names)

* **Exceptions** are errors which occur that cause the program to be unable to continue running. Code that might cause an exception can be placed in a try clause. Following the try clause is the except clause, in which one or more exceptions can be handled so that the program can recover or end in a user-friendly manner in the event of an exception being raised.

An optional else clause may follow the except clause, which runs code within this block if no exception has occurred or no break, return, or continue statements are encountered. A finally clause may follow all other clauses in the exception handling structure, which runs code in this block regardless of what happened in the other clauses.

from sys import exc\_info

import traceback

from collections import namedtuple

filesOpened = 0 # files successfully opened, processed, and closed

try:

with open("nonExistent.txt") as data:

pass

except (IOError, FileNotFoundError) as e:

print("\nexception:", e)

# ==== use exc\_info to get extra exception info ====

# type is the type of exception being handled

# value is the instance of the exception object

# traceback refers to the address in the call stack at

# which the exception occurred

ErrTuple = namedtuple('exc\_info', 'type value traceback')

exc = ErrTuple(\*exc\_info())

print("\n%s" % str(exc))

# ==== use traceback to get info regarding line numbers

# and module and function names ====

print('\ntraceback.extract\_stack:',

traceback.format\_list(traceback.extract\_stack()), end='\n', sep='\n')

print('\ntraceback.format\_stack:',

''.join(traceback.format\_stack()), end='\n', sep='\n')

else:

# no exception occurred

filesOpened += 1

print("Files successfully processed: %d" % filesOpened)

finally:

print("Finished working with file...")

* **C extensions** allow calling of C functions in Python code. The benefits of doing so include speed, low level interfacing, and working with existing C libraries. There are several ways to do so:
  + C functions can be compiled into dynamic link libraries and imported into Python using the ctypes module, which also allows for configuration of function argument and return value type adjustment.
  + Simplified Wrapper and Interface Generator is a command line utility which allows for calling of C functions in Python as well as other languages, giving this method an advantage over the previous one if multiple languages are targeted; otherwise, this method can prove cumbersome.
  + The Python/C API allows working with Python objects directly in C code, unlike the previous methods. The python.h header file contains all of the Python object types and a majority of the functions and operations necessary for working with them.

# call C code from Python

from ctypes import \*

c = CDLL('./test.dll')

c.sumFloat.argtypes = (c\_float, c\_float)

# Python assumes a C function will return either a string, int or

# bytes object so the return type here must be specified

c.sumFloat.restype = c\_float

print(c.sumFloat(c\_float(1.54782e+5), c\_float(2.58179e+9)))

print(c.sum(c\_int(10), c\_int(5)))

Project Details

One of my primary interests in the field of programming is videogame development, so throughout the semester I have worked towards developing a video game using the Python programming language. The game, entitled *Canvas Knight,* (it was originally an HTML canvas game) is an overhead action game featuring one or two players, a variety of levels, lots of monsters, and several magical spells.

Besides becoming more proficient in Python, there were several goals listed in the initial course document outlining the features I hoped to implement into the game during the semester. Those were: adding a 1-player and a 2-player mode, a scrolling feature, and additional spells and monsters. These features have since been implemented.

Implementing such features has helped me learn several fundamental programming concepts. I have learned how to:

* Use function variables and object-oriented concepts such as classes, composition and inheritance to structure, change, and understand a large program.
* Decide on which container types to use in a variety of different scenarios.
* Work with bitwise operators to improve application speed and group like data.
* Work within a multithreaded environment and alter variables across threads while properly syncing the objects on both players’ game screens.
* Serialize data and send it over network sockets using a client/server design methodology.
* Use the JSON format to create a level design scheme and design levels as well as to send and receive data over network sockets.
* Utilize modules to implement functionality that would be difficult and time consuming to program otherwise.

Below is a list of the various modules I have written which comprise the program. All of these modules rely extensively on classes to encourage abstraction and modularity while importing outside modules when necessary.

* main – the starting point of the application which moderates the order of game modes and initializes the graphics library and sets screen boundaries.
* obj – an extensive module that contains game character logic.
* utility – contains useful constants and functions other modules can use.
* ui – a simple module for working with user interface objects.
* mode – runs the various game modes during program execution.
* grid – partitions levels to reduce object rendering and update time.
* server – sends object data from the server to the client and receives and processes client input.
* client – receives object data from the server and uses it to update its game state and object rendering.

Below is a list of the various modules I imported either for the game or testing/learning purposes.

* math – contains numerous mathematical functions useful for game calculations. Canvas Knight in particular uses number truncation and floor, various trigonometric functions, hypotenuse calculation, and conversion from radians to degrees.
* json – for working with json-related functions and files. Useful for encoding and decoding data for transfer over sockets.
* random – brings unpredictability to integer and floating point values
* pygame – used to draw game graphics in Python; a wrapper for the SDL graphics library.
* collections – contains namedtuple, deque, ChainMap, Counter, and others.
* threading – for working with a multi-threaded environment.
* bottle – a micro web framework for building websites with Python.
* sqlite3 – provides a way to use SQL to work with database (.db) files in Python.
* socket – grants access to the an internal socket interface that can be used for sending data over a network.
* traceback and pdb – provide ways to retrieve stack trace information during program execution. pdb in particular contains additional functionality to aid in debugging.
* ctypes – allows calling of C functions loaded in DLLs and provides functions for interfacing with C data types.

How to Test the Application

The keyboard commands for the game are as follows:

**A**: move character left **D**: move character right

**W**: move character/menu selection up **S**: move character/menu selection down

**J**: sword attack/confirm menu selection **K**: magic

**X**: display debug information in game

**Please follow the steps below in order to test the 1-Player Mode:**

1. Open the IDLE editor.

2. Open the main.py module in the editor.

3. Press the F5 key to run the program. The program should start up and the title screen should appear.

4. Select the *1 Player* option. A level select screen will appear.

5. Select a level to begin playing the game. Levels 1-17 are levels from the original game that was written in JavaScript. *Scroll Test* (recommended) demonstrates the scrolling feature of the game. *Crowd Test* demonstrates how the game handles a large number of objects.

**Please follow the steps below in order to test the 2-Player Mode:**

There are two ways to test the 2-player mode: open two IDLE editors on the same computer or open one IDLE editor on two computers connected via LAN. Each approach will be covered below.

**How to test the 2-Player Mode on one computer:**

1. Open two separate IDLE editors.

2. Open the main.py module in both editors.

3. Press the F5 key on both of the editors. The program should start up and the title screen should appear for both of them.

4. Select the *2 Players* option on both games.

5. Select the *Host Game* option on one of the games.

6. Select the *Join Game* option on the other game. A level should appear on both game screens.

7. Alternate controlling each character by giving each game window focus.

**How to test the 2-Player Mode on two computers connected via LAN:**

1. Open one IDLE editor on each computer.

2. Open the mode.py module in both editors.

3. Go to lines (press Alt + G) 230 and 248 and change the ip address to the first computer’s IPv4 address. You can find your IPv4 address by opening a command line window and entering the command *ipconfig*.

4. Go to lines (press Alt + G) 231 and 249 and change the ip address to the second computer’s IPv4 address.

5. Save changes on both computers.

6. Open the main.py module in both editors.

7. Press the F5 key on both of the editors. The program should start up and the title screen should appear for both of them.

8. Select the *Host Game* option on one of the games.

9. Select the *Join Game* option on the other game. A level should appear on both game screens.

10. Alternate controlling each character on each computer or have another player join in!

Continued Learning Objectives

Overall, I feel I have achieved what I wanted to this semester. I plan to continue learning Python by researching, experimenting, and building applications with it. I would also like to take some time to experiment with and improve my knowledge in various web frameworks and in other programming languages such as JavaScript, PHP, and C.