Principles of computing

Week 1

(1) coding style

https://class.coursera.org/principlescomputing1-005/wiki/view?page=style\_guidelines

**Docstrings** describe **what** it is being done in a module, class, method, function, but **how** it is being done.

**comments** should describe **how** a section of code is accomplishing something.

**variables** should have a meaningful name that is at least three-characters long and should all be used. Don’t have global variables in the program except Constants with name in all caps.

**Class** name should all be in uppercase letters. **Class fields** should never be accessed directly from outside the class and start the class field names with an underscore.

(2) Python modules

import xx as xx can rename a module so that it will be easier to call

import examples\_module as example

dir(module) can print all methods of a model

Week 2

(1) Testing

Testing is important and an integral part of development

testing should be done whenever you have written a small piece of codes

using poc\_simpletest module in CodeSkupltor

e.g.

import poc\_simpletest

def run\_test(format\_function):

suite = poc\_simpletest.TestSuite() #create a test suite object

suite.run\_test(format\_function(0)), ‘0:00.0’, ‘Test #1:’)

suite.run\_test(format\_function(7)), ‘0:00.7’, ‘Test #2:’)

suite.report\_results()

(2) Plotting

use the simpleplot module in CodeSkulptor

simpleplot takes inputs of a bunches of points and plot them

e.g. http://www.codeskulptor.org/#poc\_plot\_example.py

def double(num):

"""

Example of linear function

"""

return 2 \* num

def create\_plots(begin, end, stride):

# generate x coordinates for plot points

x\_coords = []

current\_x = begin

while current\_x < end:

x\_coords.append(current\_x)

current\_x += stride

# compute list of (x, y) coordinates for each function

double\_plot = [(x\_val, double(x\_val)) for x\_val in x\_coords]

# plot the list of points

simpleplot.plot\_lines("Plots of three functions", 600, 400, "x", "f(x)",

[double\_plot, square\_plot, exp\_plot],

True, ["double", "square", "exp"])

create\_plots(0, 2, .1)

we can use plot to test the running time of different input size

to see whether it is linear or quadratic

but try to avoid exponential

(3) Grids

https://class.coursera.org/principlescomputing1-005/wiki/view?page=grids

A grid is a partition of a 2D region into a collection of disjoint cells

Mathematically, such a grid has a height- **grid\_height** and a width- **grid\_width**, measured in terms of individual squares. The standard terminology when referring to the size of such a grid is to specify height first, followed by width. For example, a three-by-four grid is three squares high and four squares wide.

We refer to a certain cell in the grid using a pair of integers- (row\_index, column\_index).

e.g. (2, 3)- 3rd row, 4th column

e.g.

http://www.codeskulptor.org/#poc\_indexed\_grid.py

http://www.codeskulptor.org/#poc\_grid\_traversal.py

create a grid with the value in each cell equals the sum of the row index and column index

EXAMPLE\_GRID = [[row + col for col in range(GRID\_WIDTH)]

for row in range(GRID\_HEIGHT)]

Week 3

(1) probability

https://class.coursera.org/principlescomputing1-005/wiki/view?page=probability

A **trial** (or an experiment) is any procedure that can be infinitely repeated and has a well-defined set of possible outcomes, known as the sample space.

An **event** is a set of outcomes of a trial (a subset of the sample space).

python for roll\_die simulation

http://www.codeskulptor.org/#poc\_basic\_probability.py

expected value can be regarded as the mean of the results when we run the trials infinite times

(2) Monte Carlo Methods

perform a bunch of random trials to figure out the expectation that some events will actually occur.

for those events difficult to compute the probability of occurrence directly

e.g.

http://www.codeskulptor.org/#examples\_monte\_carlo.py

(3) objects and references

objects

In Python, everything is an object; But in Java, this is not true

a piece of memory, holding the data

store both data and behavior

reference

variables are actually references, they’re a way to refer to objects

like the name of an object

e.g.

lst1 = [1, 2, 3]

lst 2 = lst 1

both lst1 and lst2 are references

e.g.

def func(alist):

alist[2] = 100

return alist

mylist = [‘a’, ‘b’, ‘c’]

yourlist = func(mylist)

after running this code, both mylist and yourlist refers to [‘a’, ‘b’, ‘100’]

Week 4

(1) Enumeration

https://class.coursera.org/principlescomputing1-005/wiki/view?page=enumeration

http://www.codeskulptor.org/#poc\_enumeration.py

for sequences of outcomes

case 1 : if certain outcome can be repeated

n sequence length, and m outcome set size: mn

if the sequence of outcomes is not important: only sorted sequences counts

(2) Permutations and combinations

https://class.coursera.org/principlescomputing1-005/wiki/view?page=permutations

**Permutation:**

Given a set of outcomes, a sequence of outcomes of length n with no repetitive is a permutation.

m P n = m\*(m-1)\*…\*(m-n+1) = m!/(m-n)!

**Combination:**

the order of the resulting sequence does not matter.

m C n = m! / ((m-n)!n!)

application: possible passwords

length and characters can be used increase the possible passwords

but restrictions like at least 1 number narrow down the size, meanwhile decrease predictability

窗体顶端

窗体底端

Week 5

(1) Counting and Sums

http://www.codeskulptor.org/viz/#poc\_counting.py

try to estimate the number of statement executed:

for single statement: only once

for conditions: can be estimated as the largest statements numbers in either condition

for loops: execute the condition statement and the loop body for loop times

for functions: it is decided by the inputs to the function(can be linear, quadratic, …)

Arithmetic sums

https://class.coursera.org/principlescomputing1-005/wiki/view?page=arithmetic\_sums

(2) finding the max

float(‘-inf’) negative infinity

take a list and return the max value:

method 1

def max1(data):

maxval = float(‘-inf’)

for item in data:

if item > maxval:

maxval = item

return maxval

method 2 (looks nicer, but executes exactly the same)

def max2(data):

return max(data)

method 3 (take a in list and return both the max value and its index)

def max3(data):

maxval = float(‘-inf’)

maxidx = None

for idx in range(len(data)):

if data[idx] > maxval:

maxval = data[idx]

maxidx = idx

return maxidx, maxval

method 4 (looks nicer than method 3, but in fact loop through the list twice)

def max4(data):

maxval = max(data)

maxidx = data.index(maxval)

return maxidx, maxval

method 5 (return the max values as the time it appears)

def max5(data):

maxvals = []

for item in data:

if (len(maxvals) == 0):

maxvals.append(iem)

elif item == maxvals[0]

maxvals.apend(item)

elif item > maxvals[0]:

maxvals = [item]

return maxvals

method 6 (return the indeces and the max values)

def max6(data):

maxvals = []

for idx in range(len(data)):

item = data[idx]

if (len(maxvals) == 0):

maxvals.append((idx, item))

elif item == maxvals[0][1]:

maxvals.append((idx, item))

elif item > maxvals[0][1]:

maxvals = [(idx, item)]

return maxvals

(3) Higher-order functions

functions that take other functions as arguments or can even return functions

prototype

def double(val):

return 2 \* val

def square(val):

return val \*\* 2

higher-order function

def twice(func, val):

return func(func(val))

print twice(double, 3) #12

print twice(square, 3) #81

in-built higher order functions: map()

data = [1, 3, 6, 9, 18]

newdata = map(double, data) #apply double on all elements in data to get a list

print newdata #[2, 6, 12, 18, 36]

in-built higher order functions: filter()

def even(val):

if val % 2:

return False

else:

return True

newdata2 = filter(even, data)

#apply even on all elements in data and return those element generated True

print newdata2 #[6, 18]

(4) plotting statement counts

http://www.codeskulptor.org/#poc\_mystery\_plot.py

Week 6

(1) generators

a way of programmatically producing a sequence of values in Python (produce the entire sequence at once, more efficiently)

http://www.codeskulptor.org/#examples\_generators.py

e.g.

print "max in list:", max([num \* 2 - 3 for num in range(7)])

produce first the entire list and figure out the maximum

a typical generator expression:

print "max in gen:", max(num \* 2 - 3 for num in range(7))

for generator: pass the generator to max, and as max iterate over the generator, each successive number in a sequence is produced.

both outputs are 9

e.g. generator function

def genfunc(limit):

num = 0

while num < limit:

yield num

num = num + 1

print genfunc(7) <generator object genfunc>

for a generator function, it does not use **return**, but **yield**

if there is a yield statement in one function, it is justified as a generator.

when the generator function gets to the yield keyword, it returns the value but the function’s not over. It gives the number back to whomever called it, and when you use it and go to the next iteration, it will come back right where that is and right where the yield statement is.

It does not return any value, so you have to iterate over it.

for number in genfunc(7):

print number 0 1 2 3 4 5 6

But in generator function, there can be a **return** to end an iteration.

def genfunc2(endfunc):

num = 0

while True:

if endfunc(num):

return

yield num

num = num + 1

def endfn(num):

if num == 7:

return True

return False

for number in genfunc2(endfn):

print number 0 1 2 3 4 5 6

(2) Stacks and queues

①stacks

like building blocks

stack up several blocks from bottom to top

take off from top to bottom- unstack in the opposite order

**LIFO**

→push and pop operation

②queue

like queuing up

the fist one in the queue left first

**FIFO**

→enqueue and dequeue operation

e.g. queue class

http://www.codeskulptor.org/#poc\_queue.py

(3) inheritance

one class can inherit the behavior of another class

base class → subclass

Python don’t force a type – duck typing (but this can be confusing)

e.g.

http://www.codeskulptor.org/#examples\_inheritance.py

class Base:

def hello(self):

print "hello"

def message(self, msg):

print msg

class Sub(Base):

def message(self, msg):

print "sub:", msg

obj = Sub()

obj.hello() #hello

obj.message("what's going to happen?") #sub: what's going to happen?

baseobj = Base()

baseobj.hello() #hello

baseobj.message("another message") #another message

e.g.

http://www.codeskulptor.org/#examples\_inheritance2.py

class Base:

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

self.\_number = num

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

return str(self.\_number)

class Sub(Base):

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

Base.\_\_init\_\_(self, num)

obj = Sub(42)

print obj 42

(4) grid class

https://class.coursera.org/principlescomputing2-005/wiki/view?page=fun\_growth

https://class.coursera.org/principlescomputing2-005/wiki/view?page=grids

for the simulation of colliding balls:

without grids, the number of testing is quadratic

but with grids, it’s linear

http://www.codeskulptor.org/#poc\_grid.py

using grids to do spatial search

1-D search: just traverse the line and you will eventually find what you’re looking for

But 2-D search is much more difficult

e.g. **Breadth-first search**

http://www.codeskulptor.org/#poc\_wildfire\_student.py

https://class.coursera.org/principlescomputing2-005/wiki/view?page=bfs

we have an underlying grid that stores all the yellow cells.

all orange cells are on a queue. for each step, we dequeue an element off the queue to look at it four neighbors- if there are yellow cells(on fore), those unburned neighbors (not yellow) were added to the queue, and the original orange cell turns to yellow.

while boundary is not empty:

current\_cell ← dequeue boundary

for all neighbors neighbor\_cell of current\_cell:

if neighbor\_cell is not in visited:

add neighbor\_cell to visited

enqueue neighbor\_cell onto boundary

This search method works on graphs as well.

Week 7

(1) Recursion

for problems that have common substructure

and know the answer in the last step

procedure:

to identify the base case- which can be solved directly

to identify the recursive base

take a piece of the problem and recursively solving a sub problem

http://www.codeskulptor.org/#examples\_recursion.py

e.g. sum up to n

base case: the answer of 1 when n = 1

recursive case: n + sumupto(n-1) when n>1

s(n) = 1 n=1

s(n) = n+s(n-1) n>1

def sum\_up\_to(n):

if n == 1:

# base case

return 1

else:

# recursive case

return n + sum\_up\_to(n - 1)

e.g.

def is\_palindrome(word):

if len(word) < 2:

# base case

return True

else:

# recursive case

if word[0] != word[-1]:

return False

else:

return is\_palindrome(word[1:-1])

(2) Binary search

have a list in ascending order L

and a integer K

to answer whether K is in L

http://www.codeskulptor.org/#poc\_binary\_search.py

solution 1: Linear search

for i=0 to n-1

if L[i] == K

return True

return False

but the running time is long, if double the size of the input, you double running time

solution2:

since L is an ascending list,

first find the midpoint of the list, and check whether it is K

if it is not, compare K and midpoint value, and choose the next half to find next midpoint

then repeat

BinarySerch(L,l,r,K):

# l and r are smallest and largest index of L

if l>r:

return False

mid = l + r/2 (if not integer, rounded down)

if K = mid:

return True

else:

if K < mid:

BinarySerch(L,l,mid-1,K)

else:

BinarySerch(L,mid+1,r,K)

generally speaking, you do log2(input size) operations, so if double the size of input, only do one more operation

visualize recursion:

http://www.codeskulptor.org/viz/#poc\_viz\_recursion.py

(3) Recurrences

a specific form of recursive functions- simplified recursive function, which we have the values for some base cases already given, and we express the value of function in terms of, kind of the input parameter

http://www.codeskulptor.org/#poc\_recurrence\_examples.py

https://class.coursera.org/principlescomputing2-005/wiki/view?page=recurrences

http://www.codeskulptor.org/#poc\_recurrence\_plot.py

when need to rounding, recurrence always use the floor integer

(4) Reading files

e.g. http://www.codeskulptor.org/#examples\_files.py

import urllib2

import codeskulptor

FILENAME = "examples\_files\_dracula.txt"

url = codeskulptor.file2url(FILENAME)

netfile = urllib2.urlopen(url)

data = netfile.read()

print data

print type(data)

for line in netfile.readlines():

print line

Week 8

(1) Lambda (anonymous function)

lambda parameter : expression

e.g.

g = lambda x:x+1

g(1) 2

e.g. use lambda function in high-order function

http://www.codeskulptor.org/#examples\_lambda.py

data3 = map(lambda x: x \*\* 2, data)

data4 = filter(lambda val: val % 2 == 0, data)

(2) Trees

https://class.coursera.org/principlescomputing2-005/wiki/view?page=trees

A rooted tree is a collection of nodes and edges that can be organized recursively. The tree has a root node with an associated value and a list of references to a collection of subtrees. The root nodes of these subtrees are the children of the root node for the original tree while the root node is the parent of the children. Each node in the tree should have exactly one parent with the exception of the root, which has no parent. This last condition ensures that the tree is hierarchical since each subtree of the root is then guaranteed to be disjoint from the other subtrees.

a node with one or more children: internal node

a node without children: leaf node

full binary tree: each node has either 2 or 0(if it is a leaf) children

http://www.codeskulptor.org/#poc\_tree.py

we can recursively calculate the leaves of a node, or the height of a tree

application:

evaluate an arithmetic expression

http://www.codeskulptor.org/#poc\_arith\_expression.py

arithmetic operator: interior nodes

numbers: leaves

(3) Minimax

game playing strategy- try to minimize the maximum loss

when you have some choices, look at what the opponent can do, and calculate the maximum loss for each choice, choose the minimum one

(since opponents will of course choose the move that leads to maximum loss to you)

https://class.coursera.org/principlescomputing2-005/wiki/minimax

Week 9

(1) Assertions

Assertions are a way of checking that certain conditions hold within your program.

If the assertion condition is not met, Python immediately throws an exception and stops your program.

→find unreasonable results immediately

e.g. problem sample 1

def hypotenuse(sidea, sideb): #calculate hypotenuse of a right triangle

hyp = math.sqrt(sidea + sideb)

assert hyp > sidea and hyp > sideb, “hypotenuse too short”

assert hyp < (sidea + sideb), “hypotenuse too long”

print hypotenuse(3, 4)

AssertionError: hyponuse too short (because the hyp calculation is run)

e.g. problem sample 2

def odd\_fraction(lst): #return the proportion of odd elements in a list

odds = 0

total = 0

for item in lst:

if item % 2:

odds += 1

assert total != 0, “no elements”

fraction = float(odds) / total

numbers = [2, 8, 1, 4, 3, 7, 9]

print odd\_fraction(numbers)

AssertionError: no elements (because forgot to increment total)

(2) Invariants

a logical condition that holds repeatedly at certain points during the evaluation of your program

can guide us building program

verify our program’s actually doing what we expect

https://class.coursera.org/principlescomputing2-005/wiki/view?page=invariants

fragment\_0

assert invariant(...)

fragment\_1

assert invariant(...)

...

assert invariant(...)

fragment\_n

e.g. loop invariants

def iterative\_factorial(num):

answer = 1

index = 0

assert answer == math.factorial(index)

while index < num:

index += 1

answer \*= index

assert answer == math.factorial(index)

return answer

e.g.

http://www.codeskulptor.org/#poc\_invariants.py

(3) Modeling