(U)

# Dictionary and File Exercises



0

Updated over 2 years ago by [redacted] in COMP 3321

3 317 33

fcs6 comp3321 file dictionary python exercises

(U) Dictionary and file exercises for COMP3321.

Recommendations

# Lists and Dictionary Exercises

## Exercise 1 (Euler's multiples of 3 and 5 problem)

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

```
multiples_3 = [i for i in range(3,1000,3)]
multiples_5 = [i for i in range(5,1000,5)]
multiples = set((multiples_3 + multiples_5)) # set will remove duplicate numbers
sum(multiples) # add all the numbers together
```

you can also do this in one line:

```
sum([i for i in range(3,1000) if i % 3 == 0 or i% 5 == 0])
```

## Exercise 2

Write a function that takes a list as a parameter and returns a second list composed of any objects that appear more than once in the original list

- duplicates([1,2,3,6,7,3,4,5,6]) should return [3,6]
- what should duplicates(['cow','pig','goat','horse','pig']) return?

```
# you can use a dictionary to keep track of the number of times seen
def duplicates(x):
    dup={}
    for i in x:
        dup[i] = dup.get(i,0)+1
    result = []
    for i in dup.keys():
        if dup[i] > 1:
            result.append(i)
    return result
```

```
x = [1,2,3,6,7,3,4,5,6]
duplicates(x)
```

```
#you can also just use lists...
def duplicates2(x):
  dup = []
  for i in x:
    if x.count(i) > 1 and i not in dup:
      dup.append(i)
  return dup
```

```
y = ['cow','pig','goat','horse','pig']
duplicates2(y)
```

```
z = ['2016','2015','2014']
duplicates(z)
```

# Exercise 3

Write a function that takes a portion mark as input and returns the full classification

- convert_classification('U//FOUO') should return 'UNCLASSIFIED//FOR OFICIAL USE ONLY'
- convert_classification('S//REL TO USA, FVEY') should return 'SECRET//REL TO USA, FVEY'

```
# just create a "lookup table" for potenial portion marks
full_classifications = {'U//FOUO':'UNCLASSIFIED//FOR OFFICIAL USE ONLY',
                        'C//REL TO USA, FVEY':'CONFIDENTIAL//REL TO USA, FVEY',
                        'S//REL TO USA, FVEY':'SECRET//REL TO USA, FVEY',
                        'S//SI//REL TO USA, FVEY': 'SECRET//SI//REL TO USA, FVEY',
                        'TS//REL TO USA, FVEY': 'TOP SECRET//REL TO USA, FVEY',
                        'TS//SI//REL TO USA, FVEY': 'TOP SECRET//SI//REL TO USA, FVEY'}
def convert_classification(x):
   return full_classifications.get(x,'UNKNOWN')  # look up the value for the portion mark
```

```
convert_classification('U//FOUO')
```

```
convert_classification('S//REL TO USA, FVEY')
```

```
convert_classification('C//SI')
```

# File Input/Output Exercises

These exercises build on concepts in Lesson 3 (Flow Control, e.g., for loops) and Lesson 4 (Container Data Type, e.g, dictionaries). You will use all these concepts together with reading and writing from files

# First, Get the Data

Copy the sonnet from https://urn.nsa.ic.gov/t/tx6qm and paste it into a new text file named sonnet.txt.

# Exercise 1

Write a function called file_capitalize() that takes an input file name and an output file name, then writes each word from the input file with only the first letter capitalized to the output file. Remove all punctuation except apostrophe.

```
file_capitalize('sonnet.txt', 'sonnet_caps.txt') => capitalized words written to sonnet_caps.txt
```

```
# use help('') to see what each of these string methods are doing
def capitalize(sentence):
    words = sentence.split()  # use split to split the string by spaces (i.e., words)
    new_words = [ word.strip().capitalize() for word in words ] # captialize each word
    return ' '.join(new_words) # create and return one string by combing words with ' '

def remove_punct(sentence):
    # since replace() method returns a new string, you can chain calls to the replace()
    #  method in order to remove all punctuation in one line of code
    return sentence.replace('.', '').replace(',', '').replace(':', '').replace(';', '')

def file_capitalize(infile_name, outfile_name):
    infile = open(infile_name, 'r') # open the input file
    outfile = open(outfile_name, 'w') # open the outpu file

    for line in infile: # loop through each line of input
        outfile.write(capitalize(remove_punct(line)) + '\n') # write the capitalized version to the output file

    infile.close() # finally, close the files
    outfile.close()
```

```
file_capitalize('sonnet.txt', 'sonnet_caps.txt')
```

# Exercise 2

Make a function called file_word_count() that takes a file name and returns a dictionary containing the counts for each word. Remove all punctuation except apostrophe. Lowercase all words.

```
file_word_count('sonnet.txt') => { 'it': 4, 'me': 1, ... }
```

```
def file_word_count(infile_name):
    word_counts = {}

    with open(infile_name, 'r') as infile:  # using 'with' so we don't have to close the file
        for line in infile: # loop over each line in the file
            words = remove_punct(line)  # we can use the remove_punct from exercise above
            words = words.split() # split the line into words

            for word in words: # loop over each word
                word = word.strip().lower()
                 # add one to the current count for the word (start at 0 if not there)
                word_counts[word] = word_counts.get(word, 0) + 1

    return word_counts # return the whole dictionary of word counts
```

```
counts = file_word_count('sonnet.txt')
counts
```

# Extra Credit

Write the counts dictionary to a file, one key:value per line.

```
def write_counts(outfile_name, counts):
    with open(outfile_name, 'w', encoding='utf-8') as outfile:
        # to loop over a dictionary, use the items() method
        # items() will return a 2-element tuple containing a key and a value
        # below we pull out the values from the tuple into their own variables, word and count
        for word, count in counts.items():
            outfile.write(word + ':' + str(count) + '\n') # write out in key:value format
```

```
write_counts('sonnet_counts.txt', counts)  # use the counts dictionary from Exercise 2 above
```

(U)

# Structured Data and Dates Exercise

0

(b)(3)-P.L. 86-36

Updated over 3 years ago by [redacted] in COMP 3321

3 164 13

comp3321 datetime json csv data fcs6 exercises

(U) COMP3321 exercise for working with structured data and dates.

Recommendations

# Structured Data and Dates Exercise

Save the Apple stock data from https://urn.nsa.ic.gov/t/0grli to aapl.csv.

Use DictReader to read the records. Take the daily stock data and compute the average adjusted close ("Adj Close") per week. Hint: Use .isocalendar() for your datetime object to get the week number.

For each week, print the year, month, and average adjusted close to two decimal places.

```
Year 2015, Week 23, Average Close 107.40
Year 2015, Week 22, Average Close 105.10
```

Doc ID: 6689694

```
from csv import DictReader
from datetime import datetime

def average(numbers):
    if len(numbers) == 0:
        return 0.0

    return sum(numbers) / float(len(numbers))

def get_year_week(record):
    dt = datetime.strptime(record['Date'], '%Y-%m-%d')
    return (dt.year, dt.isocalendar()[1])

def get_averages(data):
    avgs = {}

    for year_week, closes in data.items():
        avgs[year_week] = average(closes)

    return avgs

def weekly_summary(reader):
    weekly_data = {}

    for record in reader:
        year_week = get_year_week(record)

        if year_week not in weekly_data:
            weekly_data[year_week] = []

        weekly_data[year_week].append(float(record['Adj Close']))

    return get_averages(weekly_data)

def file_weekly_summary(infile_name):
    with open(infile_name, 'r') as infile:
        return weekly_summary(DictReader(infile))

def print_weekly_summary(weekly_data):
    for year_week in reversed(sorted(weekly_data.keys())):
```

```
        year = year_week[0]
        week = year_week[1]
        avg = weekly_data[year_week]
        print('Year {year}, Week {week}, Average Close {avg:.2f}'.format(year=year, week=week, avg=avg))
```

```
data = file_weekly_summary('aapl.csv')
print_weekly_summary(data)
```

# Extra

Use csv.DictWriter to write this weekly data out to a new CSV file.

```
from csv import DictWriter

def write_weekly_summary(weekly_data, outfile_name):
    headers = [ 'Year', 'Week', 'Avg' ]

    with open(outfile_name, 'w', newline='') as outfile:
        writer = DictWriter(outfile, headers )
        writer.writeheader()

        for year_week in reversed(sorted(weekly_data.keys())):
            rec = { 'Year': year_week[0], 'Week': year_week[1], 'Avg': weekly_data[year_week] }
            writer.writerow(rec)
```

```
data = file_weekly_summary('aapl.csv')
write_weekly_summary(data, 'aapl_summary.csv')
```

# Extra Extra

Use json.dumps() to write a JSON entry for each week on a new line.

```
import json

def write_json_weekly_summary(weekly_data, outfile_name):
    with open(outfile_name, 'w') as outfile:
        for year_week in reversed(sorted(weekly_data.keys())):
            rec = { 'year': year_week[0], 'week': year_week[1], 'avg': weekly_data[year_week] }
            outfile.write(json.dumps(rec) + '\n')
```

Doc ID: 6689694

```
data = file_weekly_summary('aapl.csv')
write_json_weekly_summary(data, 'aapl.json')
```

(U)

# Datetime Exercise Solutions

(b)(3)-P.L. 86-36

0

Created almost 3 years ago by [redacted] in COMP 3321

3 1 143 17

comp3321 exercises

(U) Solutions for the Datetime exercises

Recommendations

# (U) Datetime Exercises

(U) How long before Christmas?

```
import datetime, time

print(datetime.date(2017, 12, 25) - datetime.date.today())
```

***(U) Or, if you're counting the microseconds:***

```
print(datetime.datetime(2017, 12, 25) - datetime.datetime.today())
```

(U) How many seconds since you were born?

```
birthdate = datetime.datetime(1985, 1, 31)
time_since_birth = datetime.datetime.today() - birthdate
print('{:,}'.format(time_since_birth.total_seconds()))
```

(U) What is the average number of days between Easter and Christmas for the years 2000 - 2999?

```
from dateutil.easter import easter
total = 0
span = range(2000, 3000)
for year in span:
    total += (datetime.date(year, 12, 25) - easter(year)).days

average = total / len(span)
print('{:6.4f}'.format(average))
```

(U) What day of the week does Christmas fall on this year?

```
datetime.date(2015, 12, 25).strftime('%A')
```

(U) You get a intercepted email with a POSIX timestamp of 1435074325. The email is from the leader of a Zendian extremist group and says that there will be an attack on the Zendian capitol in 14 hours. In Zendian local time, when will the attack occur? (Assume Zendia is in the same time zone as Kabul)

```
import pytz
utc_tz = pytz.timezone('Etc/UTC')
email_time_utc = datetime.datetime.fromtimestamp(1435074325, tz=utc_tz)
attack_time_utc = email_time_utc + datetime.timedelta(hours=14)
zendia_tz = pytz.timezone('Asia/Kabul')
attack_time_zendia = attack_time_utc.astimezone(zendia_tz)

print(email_time_utc)
print(attack_time_utc)
print(attack_time_zendia)
```

(U)

# Object Oriented Programming and Exceptions Exercise

(b)(3)-P.L. 86-36

0

Created over 3 years ago by [redacted] in COMP 3321

3 2 125 15

fcs6 oop comp3321 exceptions objects classes exercises

(U) COMP3321 exercise for object oriented programming and exceptions.

Recommendations

## Object Oriented Programming and Exceptions Exercise

Make a class called Symbol that holds data for a stock symbol, with the following properties:

```
self.name
self.daily_data
```

It should also have the following functions:

```
def __init__(self, name, input_file)
def data_for_date(self, date_str)
```

**init**(self, name, input_file) should open the input file and read it with DictReader, putting each entry in self.daily_data, using the date strings as the keys. Make sure to open the daily data file within a try/except block in case the file does not exist. If the file does not exist, set self.daily_data to an empty dictionary.

data_for_date(self, date_str) should take a date string and return the dictionary containing that days' data. If there is no entry for that date, return an empty dictionary.

Doc ID: 6689694

# Tests

Make sure the following execute as specified in each comment. You can get the aapl.csv file from https://urn.nsa.ic.gov/t/0grli. The apple.csv file should not exist.

```
s1 = Symbol('AAPL', 'aapl.csv')
print(s1.data_for_date('2015-08-10'))  # should return a dictionary for that date
print(s1.data_for_date('2015-08-09'))  # should return an empty dictionary

s2 = Symbol('AAPL', 'apple.csv')       # should not raise an exception!
print(s2.data_for_date('2015-08-10'))  # should return an empty dictionary
print(s2.data_for_date('2015-08-09'))  # should return an empty dictionary
```

# Module: Collections and Itertools



0

Updated almost 2 years ago by [redacted] in COMP 3321

3 1 296 110

fcs6 python

(U) Module: Collections and Itertools

Recommendations

(U) Any programming language has to strike a balance between the number of basic elements it exposes, like control structures, data types, and so forth, and the utility of each one. For example, Python could do without `tuple` s entirely, and could replace the `dict` with a `list` of `list` s or even a single `list` where even-numbered indices contain *keys* and odd-numbered indices contain *values*. Often, there are situations that happen so commonly that they warrant inclusion, but inclusion in the **builtin** library is not quite justified. Such is the case with the **collections** and **itertools** modules. Many programs could be simplified with a `defaultdict` , and having one available with a single `from collection import defaultdict` is much better than reinventing the wheel every time it's needed.

## (U) Value Added Containers with **collections**

(U) Suppose we want to build an index for a poem, so that we can look up the lines where each word occurs. To do this, we plan to construct a dictionary with the words as keys, and a list of line numbers is the value. Using a regular `dict` , we'd probably do something like this:

```
poem = """mary had a little lamb
it's fleece was white as snow
and everywhere that mary went
the lamb was sure to go"""
```

```
index = {}
```

```
for linenum, line in enumerate(poem.split('\n')):
    for word in line.split():
        if word in index:
            index[word].append(linenum)
        else:
            index[word] = [linenum]
```

(U) This code would be simpler without the inner `if ... else ...` clause. That's exactly what a `defaultdict` is for; it takes a function (often a `type`, which is called as a constructor without arguments) as its first argument, and calls that function to create a *default* value whenever the program tries to access a key that isn't currently in the dictionary. (It does this by overriding the `__missing__` method of `dict`.) In action, it looks like this:

```
from collections import defaultdict
```

```
index = defaultdict(list)
```

```
for linenum, line in enumerate(poem.split('\n')):
    for word in line.split():
        index[word].append(linenum)
```

(U) Although a `defaultdict` is almost exactly like a dictionary, there are some possible complications because it is possible to add keys to the dictionary unintentionally, such as when testing for membership. These complications can be mitigated with the `get` method and the `in` operator.

```
'sheep' in index # False
```

```
1 in index.get('sheep') # Error
```

```
'sheep' in index # still False
```

```
2 in index['sheep'] # still False, but...
```

```
'sheep' in index # previous statement accidentally added 'sheep'
```

(U) You can do crazy things like change the `default_factory` (it's just an attribute of the `defaultdict` object), but it's not commonly used:

```
import itertools
```

```
def constant_factory(value):
    return itertools.repeat(value).__next__
```

```
d = defaultdict(constant_factory('<missing>'))
```

```
d.update(name='John', action='ran')
```

```
'{0[name]} {0[action]} to {0[object]}'.format(d)
```

```
d # "object" added to d
```

(U) A `Counter` is like a `defaultdict(int)` with additional features. If given a `list` or other iterable when constructed, it will create counts of all the unique elements it sees. It can also be constructed from a dictionary with numeric values. It has a custom implementation of `update` and some specialized methods, like `most_common` and `subtract`.

```
from collections import Counter
```

```
word_counts = Counter(poem.split())
```

```
word_counts.most_common(3)
```

```
word_counts.update('lamb lamb lamb stew'.split())
```

```
word_counts.most_common(3)
```

```
c = Counter(a=3, b=1)
```

```
d = Counter(a=1, b=2)
```

```
c + d
```

```
c - d          # Did you get the output you expected?
```

```
(c - d) + d
```

```
c & d
```

```
c | d
```

(U) An `OrderedDict` is a dictionary that remembers the order in which keys were originally inserted, which determines the order for its iteration. Aside from that, it has a `popitem` method that can pop from either the beginning or end of the ordering.

(U) `namedtuple` is used to create lightweight objects that are somewhat like tuples, in that they are immutable and attributes can be accessed with `[]` notation. As the name indicates, attributes are named, and can also be accessed with the `.` notation. It is most often used as an optimization, when speed or memory requirements dictate that a `dict` or custom object isn't good enough. Construction of a `namedtuple` is somewhat indirect, as `namedtuple` takes field specifications as strings and returns a `type`, which is then used to create the named tuples. named tuples can also enhance code readability.

```
from collections import namedtuple
```

```
Person = namedtuple('Person', 'name age gender')

bob = Person(name='Bob', age=30, gender='male')

print( '%s is a %d year-old %s' % bob )  # 2.x style string formatting

print( '{} is a {} year-old {}'.format(*bob) )

print( '%s is a %d year-old %s' % (bob.name, bob.age, bob.gender) )

print( '{} is a {} year-old {}'.format(bob.name, bob.age, bob.gender) )

bob[0]

bob['name']                  # TypeError

bob.name

print( '%(name)s is a %(age)d year-old %(gender)s' % bob )    # Doesn't work

print( '{name} is a {age} year-old {gender}'.format(*bob) )  # Doesn't work

print( '{0.name} is a {0.age} year-old {0.gender}'.format(bob) )  # Works!
```

(U) Finally, `deque` provides queue operations.

```
from collections import deque

d = deque('ghi')        # make a new deque with three items

d.append('j')           # add a new entry to the right side

d.appendleft('f')       # add a new entry to the left side

d.popleft()             # return and remove the leftmost item

d.rotate(1)             # right rotation

d.extendleft('abc')     # extendleft() reverses the input order
```

(U) The **collections** module also provides Abstract Base classes for common Python interfaces. Their purpose and use is currently beyond the scope of this course, but the documentation is reasonably good.

# (U) Slicing and Dicing with **itertools**

Given one or more `list` s, `iterator` s, or other iterable objects, there are many ways to slice and dice the constituent elements.
The **itertools** module tries to expose building block methods to make this easy, but also tries to make sure that its methods are useful in a variety of situations, so the documentation contains a cookbook of common use cases. We only have time to cover a small subset of the **itertools** functionality. Methods from **itertools** usually return an iterator, which is great for use in loops and list comprehensions, but not so good for inspection; in the code blocks that follow, we often call `list` on these things to unwrap them.

(U) The `chain` method combines iterables into one super-iterable. The `groupby` method separates one iterator into groups of adjacent objects, possibly as determined by an optional argument–this can be tricky, especially because there's no look back to see if a new key has been encountered previously.

```
import itertools
```

```
list(itertools.chain(range(5),[5,6])) == [0,1,2,3,4,5,6]
```

```
size_groups = itertools.groupby([1,1,2,2,2,'p','p',3,4,3,3,2])
```

```
[(key, list(vals)) for key, vals in size_groups]
```

(U) A deeply nested for loop or list comprehension might be better served by some of the *combinatoric generators* like `product` , `permutations` , or `combinations` .

```
iter_product = itertools.product([1,2,3],['a','b','c'])
```

```
list(iter_product)
```

```
iter_combi = itertools.combinations("abcd",3)
```

```
list_combi = list(iter_combi)
list_combi
```

```
iter_permutations = itertools.permutations("abcd",3)
```

```
list(iter_permutations)
```

(U) `itertools` can also be used to create generators:

```
counter = itertools.count(0, 5)
```

```
next(counter)
```

```
print(list(next(counter) for c in range(6)))
```

(U) Be careful... What's going on here?!?

```
counter = itertools.count(0.2,0.1)
for c in counter:
    print(c)
    if c > 1.5:
        break
```

```
cycle = itertools.cycle('ABCDE')
```

```
for i in range(10):
    print(next(cycle))
```

```
repeat = itertools.repeat('again!')
```

```
for i in range(5):
    print(next(repeat))
```

```
repeat = itertools.repeat('again!', 3)
for i in range(5):
    print(next(repeat))
```

```
nums = range(10,0,-1)
my_zip = zip(nums, itertools.repeat('p'))
for thing in my_zip:
    print(thing)
```

# Functional Programming

(b)(3)-P.L. 86-36

0

Created over 3 years ago by [redacted] in COMP 3321

3 2 193 38

fcs6 functional map python reduce

(U//~~FOUO~~) A short adaptation of [redacted] "A practical introduction to functional programming" in Python to supplement COMP 3321 materials. Also discusses lambdas.

Recommendations

UNCLASSIFIED

## (U) Introduction

(U) At a basic level, there are two fundamental programming styles or paradigms:

- *imperative* or *procedural* programming and
- *declarative* or *functional* programming.

(U) Imperative programming focuses on telling a computer how to change a program's *state*--its stored information--step by step. Most programmers start out learning and using this style. It's a natural outgrowth of the way the computer actually works. These instructions can be organized into functions/procedures (*procedural* programming) and objects (*object-oriented*programming), but those stylistic improvements remain imperative at heart.

(U) Declarative programming, on the other hand, focuses on expressing *what* the program should do, not necessarily *how* it should be done. *Functional programming* is the most common flavor of that. It treats a program as if it is made up of *mathematical*-style functions: for a given input `x`, running it through function `f` will always give you the same output `f(x)`, and `x` itself will remain unchanged afterwards. (Note that this is *not* necessarily the same as a procedural-style function, which may have access to global variables or other "inputs" and which may be able to modify those inputs directly.)

# (U) TL;DR

(U) The key distinction between procedural and functional programming is this: **a procedural function may have side effects**--it may change the state of its inputs or something outside itself, giving you a different result when running it a second time. **Functional programming avoids side effects**, ensuring that functions don't modify anything outside themselves.

# (U) Note



(U) The contents of this notebook have been borrowed from the beginning of [redacted] essay, "A practical introduction to functional programming." A full notebook version of that essay can be found here. (Note that it uses Python 2.)

# (U) Functional vs. Not

(U) The best way to understand side effects is with an example.

(U) This function is *not* functional:

```
a = 0
def increment():
    global a
    a += 1
```

(U) This function *is* functional:

```
def increment(a):
    return a + 1
```

# (U) Map-Reduce

(U) Let's jump into functional coding. One common use is **map-reduce**, which you may have heard of. Let's see if we can make sense of it.

# (U) map

(U) Conceptually, **map** is a function that takes two arguments: another function and a collection of items. It will

1. run the function on each item of the original collection and
2. return a new collection containing the results,

3. leaving the original collection unchanged.

(U) In Python 3, the input collection must simply be *iterable* (e.g. `list`, `tuple`, `string`). Its `map` function returns an *iterator* that runs the input function on each item of iterable.

# (U) Example 1: Name Lengths

(U) Take a list of names and get a list of the name lengths:

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])
print(list(name_lengths))
```

# (U) Example 2: Squaring

(U) Square every number in a list:

```
squares = map(lambda x: x * x, [0, 1, 2, 3, 4])
print(list(squares))
```

# (U) A digression on `lambda`

(U) So what's going on with that input function? `lambda` will let you define and use an *unnamed* function. Arguments fit between the `lambda` and the colon while the stuff after the colon gets *implicitly* returned (i.e. without explicitly using a `return` statement).

(U) Lambdas are most useful when:

- your function is simple and
- you only need to use it once.

(U) Consider the usual way of defining a function:

```
def square(x):
    return x * x
```

```
square(4)
```

```
# we could have done this instead
squares = map(square, [0, 1, 2, 3, 4])
print(list(squares))
```

(U) Now let's define the same function using a lambda:

```
lambda x: x * x
```

(U) Fine, but how do we call that resulting function? Unfortunately, it's too late now; we didn't store the result, so it's lost in the ether.

(U) Let's try again:

```
ima_function_variable = lambda x: x * x
```

```
type(ima_function_variable)
```

```
ima_function_variable(4)
```

```
# be careful!
ima_function_variable = 'something else'
```

```
# our lambda function is gone again
ima_function_variable(4)
```

# (U) Example 3: Code Names

(U) OK, back to `map`. Here's a procedural way to take a list of real names and replace them with randomly assigned code names.

```
import random
```

```
names = ['Mary', 'Isla', 'Sam']
code_names = ['Mr. Pink', 'Mr. Orange', 'Mr. Blonde']

for i in range(len(names)):
    names[i] = random.choice(code_names)

print(names)
```

(U) Here's the functional version:

```
names = ['Mary', 'Isla', 'Sam']
covernames = map(lambda x: random.choice(['Mr. Pink', 'Mr. Orange', 'Mr. Blonde']), names)
print(list(covernames))
```

# (U) Exercise: Code Names...Improved?

(U) The procedural code below generates code names using a new method. Rewrite it using `map`.

```
names = ['Mary', 'Isla', 'Sam']

for i in range(len(names)):
    names[i] = hash(names[i])

print(names)
```

```
# your code here
```

# reduce

(U) **Reduce** is the follow-on counterpart to map. Given a function and a collection of items, it uses the function to combine them into a single value and returns that result.

(U) The function passed to `reduce` has some restrictions, though. It must take two arguments: an *accumulator* and an *update* value. The update value is like it was before with `map`; it will get set to each item in the collection one by one. The accumulator is new. It will receive the output from the previous function call, thus "accumulating" the combined value from item to item through the collection.

(U) **Note**: in Python 2, `reduce` was a built-in function. Python 3 moved it into the `functools` package.

## (U) Example

(U) Get the sum of all items in a collection.

```
import functools
```

```
sum = functools.reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print(sum)
```

# Recursion Examples

(b)(3)-P.L. 86-36

0

Updated over 3 years ago by [redacted]

3 193 46

python examples recursion fcs6 fibonacci games ai

(U) Some simple recursion examples in Python

Recommendations

# Recursion

Recursion provides a way to loop without loops. By calling itself on updated data, a recursive function can progress through a problem and traverse the options.

# Nth Fibonacci Number

https://wikipedia.nsa.ic.gov/en/Fibonacci_number

This returns the nth Fibonacci number in the Fibonacci Sequence using recursion.

```
def nth_fibonacci(n):
    if n < 1:
        return 0
    elif n == 1:
        return 1
    elif n == 2:
        return 1
    else:
        return nth_fibonacci(n-2) + nth_fibonacci(n-1)
```

```
nth_fibonacci(10)
```

# Fibonacci Sequence

This returns a list of the first n Fibonacci Numbers using recursion.

```
def fibonacci(n, seq=[]):
    if len(seq) == n:
        return seq
    elif len(seq) == 0:
        return fibonacci(n, [1])
    elif len(seq) == 1:
        return fibonacci(n, [1,1])
    else:
        next_value = seq[-2] + seq[-1]
        return fibonacci(n, seq + [next_value])
```

```
fibonacci(5)
```

# Simple Game

This simple game just takes in a list of nine elements and tries to modify each slot until all the numbers from 1 to 9 are in the list.

```
import random
```

```
def improve(input_list, missing):
    random_index = random.choice(list(range(len(input_list))))
    random_value = random.choice(missing)
    new_list = input_list[:]
    new_list[random_index] = random_value
    return new_list

def find_missing(input_list):
    missing = [ x for x in list(range(1,10)) if x not in input_list ]
    return missing

def one_to_nine(input_list):
    print(input_list)
    missing = find_missing(input_list)

    if len(missing) == 0:
        return input_list
    else:
        new_list = improve(input_list, missing)
        return one_to_nine(new_list)
```

```
one_to_nine([1,1,1,1,1,1,1,1,1])
```

# Simple Game Revised

This revision of the same simple game comes up with a list of possible improvements and tries to pick the best one to pursue. You'll notice that it takes fewer attempts to reach an anser than the original version of this simple game.

Doc ID: 6689695

```
def improve(input_list, missing):
    random_index = random.choice(list(range(len(input_list))))
    random_value = random.choice(missing)
    new_list = input_list[:]
    new_list[random_index] = random_value
    return new_list

def find_missing(input_list):
    missing = [ x for x in list(range(1,10)) if x not in input_list ]
    return missing

def score(input_list):
    missing = find_missing(input_list)
    return (len(missing), input_list)

def best_scoring_list(scored_lists):
    lowest = 100
    best_list = []

    for x in scored_lists:
        score = x[0]
        input_list = x[1]

        if score < lowest:
            lowest = score
            best_list = input_list

    return best_list

def one_to_nine(input_list):
    print(input_list)
    missing = find_missing(input_list)

    if len(missing) == 0:
        return input_list
    else:
        possible_improvements = [ improve(input_list, missing) for i in range(len(missing)) ]
        scored_improvements = [ score(i) for i in possible_improvements ]
        best_list = best_scoring_list(scored_improvements)
        return one_to_nine(best_list)
```

```
one_to_nine([1,1,1,1,1,1,1,1,1])
```

# Simple Game as a Tree

We can think of our strategy as a tree of options that we traverse, following branches that show that they're going to improve our chances of finding a solution. This is an extremely simplified form of what many video games use for their AI. We put our possible_improvements in a generator so they will only be created as needed. If we put them in a list comprehension as before, then all possible improvements would be generated even though many of them will likely go unused. In the end, we return any() with a generator for the branches. Since any() only needs one item to be True, it will return True as soon as a solution is found; when len(missing) == 0.

You'll notice that this results in more iterations of one_to_nine() than in the previous revision. However, the previous revision also generated a lot of data that ends up getting discarded. In other words, there's probably more processing and memory consumed by the previous revision behind the scenes.

```
def improve(input_list, missing):
    random_index = random.choice(list(range(len(input_list))))
    random_value = random.choice(missing)
    new_list = input_list[:]
    new_list[random_index] = random_value
    return new_list

def find_missing(input_list):
    missing = [ x for x in list(range(1,10)) if x not in input_list ]
    return missing

def one_to_nine(input_list, prev_missing=None):
    print(input_list)
    missing = find_missing(input_list)

    if prev_missing is None:
        return one_to_nine(input_list, missing)
    elif len(missing) == 0:
        return True
    elif len(missing) > len(prev_missing):
        return False
    else:
        possible_improvements = ( improve(input_list, missing) for i in range(len(missing)) )
        return any(( one_to_nine(p, missing) for p in possible_improvements ))
```

Doc ID: 6689695

```
one_to_nine([1,1,1,1,1,1,1,1,1])
```

# Module: Command Line Arguments

0

(b)(3)-P.L. 86-36

Updated almost 2 years ago by [redacted] in COMP 3321

3 323 123

python

(U) Module: Command Line Arguments

Recommendations

## UNCLASSIFIED

(U) Most command line programs accept options and arguments, and many even provide help messages that indicate what options are available, and how they are to be used. For example, the utility program `mv` takes two arguments, and most often moves the first argument (the source) to the second (the destination). It has other ways of operating, which are enabled by optional flags and arguments; from a command prompt, type `mv --help` to see more.

(U) There are several ways to enable this type of functionality in a Python program, and the best way to do it has been a source of contention. In particular, this lesson will cover the **argparse**module, which was added to the standard library in Python 2.7, and not the **optparse** module which was deprecated at that time.

(U) Everything passed as arguments to a Python program is available in the interpreter as the list of strings in `sys.argv` . In an interactive session, `sys.argv` always starts out as `['']` . When running a script, `sys.argv[0]` is the name of the script. We start by examining what `sys.argv` looks like. Put the following commands in a file called `argtest.py` or similar:

```
import sys
print(sys.argv)
```

```
# ...or make python do it!
contents = '''import sys
print(sys.argv)
'''

with open('argtest.py', 'w') as f:
    f.write(contents)
```

(U) Close the file and execute it from the command line with some arguments:

```
# the '!' at the beginning tells jupyter to send what follows to the command line
!python3 argtest.py -xzf --v foo --othervar=bar file1 file2
```

```
# => ['argtest.py', '-xzf', '--v', 'foo', '--othervar=bar', 'file1', 'file2']
```

(U) In all of the argument parsing that follows, `sys.argv` will be involved, although that may happen either implicitly or explicitly. Although it is often unwise to do so within a script, `sys.argv` can be modified, for instance during testing within an interactive session.

(U) Note that in Jupyter you still have `argv`, but it may not be what you expect. If you look at it, you'll see how this Python 3 kernel is being called:

```
import sys
print(sys.argv)
```

# (U) The Hard Way: **getopt**

(U) For programs with only simple arguments, the **getopt** module provides functionality similar to the `getopt` function in C. The main method in the module is `getopt`, which takes a list of strings, usually `sys.argv[1:]` and parses it according to a string of options, with optional *long options*, which are allowed to have more than one letter; explanations are best left to examples. This method returns a pair of lists, one containing `(option, value)` tuples, the other containing additional positional arguments. These values must then be further processed within the program; it might be useful, for instance, to put the `(option, value)` tuples into a `dict`. If `getopt` receives an unexpected option, it throws an error. If it does not receive all the arguments it requests, no error is thrown, and the missing arguments are not present in the returned value.

```
import getopt
```

```
getopt.getopt('-a arg'.split(),'a:') # a expects an argument
```

```
getopt.getopt('-a arg'.split(), 'a:b') # no b, no problem
```

```
getopt.getopt('-b arg -a my-file.txt'.split(), 'ab:') # my-file.txt is argument, not option
```

```
getopt.getopt('-a arg --output=other-file.txt my-file.txt'.split(),'a:b',['output=']) # long options
```

(U) For programs that use `getopt`, usage help must be provided manually.

```
def usage():
    print("""usage: my_program.py -[abh] file1, file2, ...""")
```

```
# this won't actually find anything in Jupyter, since ipython3 probably doesn't have these options
opts, args = getopt.getopt(sys.argv[1:], 'abh')
```

```
opt_dict = dict(opts)
```

```
if '-h' in opt_dict:
    usage()
```

# (U) The **argparse** Module

(U) Integrated help is one of the benefits of **argparse**, along with the ability to specify both short and long options versions of arguments. There is some additional complication in setting up **argparse**, but it is the right thing to do for all but the most simple programs.

## (U) Basic Usage

(U) The main class in **argparse** is `ArgumentParser`. After an `ArgumentParser` is instantiated, arguments are added to it with the `add_argument` method. After all the arguments are added, the `parse_args` method is called. By default, it reads from `sys.argv[1:]`, but can also be passed a list of strings, primarily for testing. Both positional arguments (required) and optional arguments indicated by flags are supported. An example will illustrate the operation.

```
import argparse
```

```
parser = argparse.ArgumentParser()
```

```
parser.add_argument('n')
```

```
parser.add_argument('-f')
```

```
parser.add_argument('-i','--input')
```

```
parser.print_help()
```

```
parser.parse_args('abc -f xyz'.split())
```

```
parser.parse_args('-f xyz abc --input=myfile.txt'.split())
```

```
parser.parse_known_args('-f xyz abc --input=myfile.txt -o otherfile.txt'.split())
```

```
args = parser.parse_args('-f xyz abc --input=myfile.txt'.split())
```

```
args.f
```

(U) As seen in the final two lines, positional arguments and optioned arguments can come in any order, which is not the case with **getopt**. If multiple positional arguments are specified, they are parsed in the order in which they were added to the `ArgumentParser`. The object returned by `parse_args` is called a `Namespace`, but it is just an object which contains all the parsed data. Unless otherwise specified, the attribute names are derived from the option names. Positional arguments are used directly, while short and long flags have leading hypens stripped and internal hyphens converted to underscores. If more than one flag is specified for an argument, the first long flag is used if present; otherwise, the first short flag is used.

(U) Here is how argparse could look in your code.

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('n')
parser.add_argument('-f')
parser.add_argument('-i','--input')
args = parser.parse_args()

print(args)```
```

```
contents = '''import argparse
parser = argparse.ArgumentParser()
parser.add_argument('n')
parser.add_argument('-f')
parser.add_argument('-i','--input')
args = parser.parse_args()

print(args)
'''

with open('argparsetest.py', 'w') as f:
    f.write(contents)
```

(U) Now we can simulate running it:

```
!python3 argparsetest.py -f xyz abc --input=myfile.txt
```

```
!python3 argparsetest.py -h
```

# (U) Advanced Options

(U) The `add_argument` method supports a large number of keyword arguments that configure behavior more finely than the defaults. For instance, the `type` argument will make the parser attempt to convert arguments to the specified type, such as `int`, `float`, or `file`. In fact, you could use any class for the type, as long as it can be constructed from a single string argument.

```
parser = argparse.ArgumentParser()
```

```
parser.add_argument('n', type=int)
```

```
parser.parse_args('5'.split())
```

(U) The `nargs` keyword lets an argument specify a fixed or variable number of arguments to consume, which are then stored into a list. This applies to both positional and optional arugments. Giving `nargs` the value `'?'` makes positional arguments optional, in which case the `default` keyword is required.

```
parser = argparse.ArgumentParser()
```

```
parser.add_argument('n', nargs=2)
```

```
parser.add_argument('-m', nargs='*') # arbitrary arguments
```

```
parser.parse_args('n1 n2 -m a b c'.split())
```

```
parser.add_argument('o', nargs='?', default='OoO')
```

```
parser.parse_args('n1 n2 oOo -m a b c'.split())
```

(U) The `default` keyword can also be used with optional arguments. When an optional argument is always used as as a flag without parameters, it is also possible to use the `action='store_const'` and `const` keywords. In this case, when the option is detected, the `Namespace` is given an appropriately-named attribute with `const` as its value. If the option is not present in the parsed args, the attribute is created with the value given in `default`, or `None` if `default` isn't set.

```
parser = argparse.ArgumentParser()
```

```
parser.add_argument('-n', action='store_const', const=7)
```

```
parser.add_argument('-b', action='store_true')
```

```
parser.add_argument('-c', action='store_const', const=5, default=3)
```

```
parser.parse_args([])
```

```
parser.parse_args('-n -b'.split())
```

```
parser.parse_args('-cbn'.split())
```

(U) The `action` keyword can take other arguments; for instance, `action='store_true'` and `action='store_false'` can be used instead of setting `const` to a boolean value.

(U) Once again, we have only scratched the surface of a module, **argparse** in this case. Check out the documentation for more details (e.g. changing attribute names with the `dest` keyword, writing custom action functions, providing different parsers for subprograms).

# Module: Dates and Times

(b)(3)-P.L. 86-36

0

Updated 10 months ago by [redacted] in COMP 3321

3 2 450 204

fcs6 python datetime arrow parsedate

(U) How to manipulate dates and times in Python.

Recommendations

UNCLASSIFIED

## (U) Introduction

(U) There are many great built-in tools for date and time manipulation in Python. They are spread over a few different modules, which is a little annoying.

(U) That being said, the **datetime** module is very complete and the most useful, so we will concentrate on that one that most. The other one that has some nice methods is the **time** module.

## (U) `time` Module

(U) The **time** module is handy for its time accessor functions and `sleep` command. It is most useful when you want quick *access* to the time but don't need to *manipulate* it.

```
import time
```

```
time.time()  # "epoch" time (seconds since Jan. 1st, 1970)
```

```
time.gmtime()
```

```
time.localtime()
```

```
time.gmtime() == time.localtime()
```

```
time.asctime()  # will take an optional timestamp
```

```
time.strftime('%c')  # many formatting options here
```

```
time.strptime('Tue Nov 19 07:04:38 2013')
```

(U) The last method you might use from the time module is `sleep`. (Doesn't this seem out of place?)

```
time.sleep(10)   # argument is a number of seconds
print("I'm awake!")
```

# (U) `datetime` Module

(U) The **datetime** module is a more robust module for dates and times that is object-oriented and has a notion of datetime arithmetic.

(U) There are 5 basic types that comes with the datetime module. These are:

1. `date` : a type to store the date (year, month, day) using the current Gregorian calendar,
2. `time` : a type to store the time (hour, minute, second, microsecond, tzinfo--all idealized, with no notion of leap seconds),
3. `datetime` : a type to store both date and time together,
4. `timedelta` : a type to store the duration or difference between two date, time, or datetime instances, and
5. `tzinfo` : a base class for storing and using time zone information (we will not look at this).

# (U) `date` type

```
import datetime
```

```
datetime.date(2013, 11, 19)
```

```
datetime.date.today()
```

```
datetime.date.fromtimestamp(time.time())
```

```
today = datetime.date.today()
```

```
today.day
```

```
today.month
```

```
today.timetuple()
```

```
today.weekday()   # 0 ordered starting from Monday
```

```
today.isoweekday() # 1 ordered starting from Monday
```

```
print(today)
```

```
today.strftime('%d %m %Y')
```

# (U) time type

```
t = datetime.time(8, 30, 50, 0)
```

```
t.hour = 9
```

```
t.hour
```

```
t.minute
```

```
t.second
```

```
print(t)
```

```
print(t.replace(hour=12))
```

```
t.hour = 8
print(t)
```

# (U) datetime type

```
dt = datetime.datetime(2013, 11, 19, 8, 30, 50, 0)
```

```
print(dt)
```

```
datetime.datetime.fromtimestamp(time.time())
```

```
now = datetime.datetime.now()
```

(U) We can break apart the `datetime` object into `date` and `time` objects. We can also combine `date` and `time` objects into one `datetime` object.

```
now.date()
```

```
print(now.date())
```

```
print(now.time())
```

```
day = datetime.date(2011, 12, 30)
```

```
t = datetime.time(2, 30, 38)
```

```
day
```

```
t
```

```
dt = datetime.datetime.combine(day,t)
```

```
print(dt)
```

# (U) timedelta type

(U) The best part of the **datetime** module is date arithmetic. What do you get when you subtract two dates?

```
day1 = datetime.datetime(2013, 10, 30)
```

```
day2 = datetime.datetime(2013, 9, 20)
```

```
day1 - day2
```

```
print(day1 - day2)
```

```
print(day2 - day1)
```

```
print(day1 + day2)  # Of course not...that doesn't make sense :)
```