COMMERCE DATA SERVICE

Introduction to Python

Star Ying
dataacademy@doc.gov

Based on Google’s Python Class

https://developers.google.com/edu/python/

COMMERCE DATA SERVICE

Background

Python is a popular, object-oriented, and structured programming language. It’s design is guided by the ‘Zen of Python’ software principles.

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one— and preferably only one —obvious way to do it although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than right now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea—let's do more of those!

Extending Python

Python is highly extensible. Besides the core functionality of Python, **packages** for Python can be **import**ed to obtain additional functionality through **conda** or **pip**.

You should have installed the following packages prior to class:

- ujson
- numpy
- scipy

COMMERCE DATA SERVICE

Editing Python

A Python program is just a text file that you edit directly. The program typically is saved with a **.py** extension to differentiate. Using an editor designed for programming is preferred. **Do not use Wordpad or Notepad**.

You should have one of the following installed prior to class:

- Sublime
- Atom
- Notepad++

COMMERCE DATA SERVICE

Python Interpreter

The Python interpreter can be called by itself. This can allow you to experiment with commands and syntax as you write your program.

To call the Python interpreter itself:

python

To pass a Python program to the interpreter:

python program.py

Hello World

```
print ‘Hello World’
```

Hello World

```
print ‘Hello World’

 # But not goodbye
```

COMMERCE DATA SERVICE

Indentation Matters

Whitespace indentation of a piece of code affects its meaning. A logical block of statements should all have the same indentation. If one of the lines in a group has a different indentation, it is flagged as a syntax error.

Avoid using TABs as they greatly complicate the indentation scheme. Set your editor to insert spaces instead of TABs for Python code.

Exercise

Create a Python program, that when passed to the Python interpreter prints ‘Hello World’?

COMMERCE DATA SERVICE

Indentation Matters

```
def hello():
  print ‘Hello World’
  print ‘Hello Again’
hello()
```

Strings

Python has a built-in string class named "str". Strings can be enclosed by either double or single quotes. A double quoted string literal can contain single quotes without any fuss and likewise single quoted string can contain double quotes.

```
‘Hello World’

“Hello World’

‘“Hello World”’
```

Strings

Python strings are "immutable" which means they cannot be changed after they are created. Since strings can't be changed, we construct *new* strings as we go to represent computed values.

```
print ‘Hello’ + ’World’
```

Strings

Characters in a string can be accessed using the standard [ ] syntax. If the index is out of bounds for the string, Python raises an error. The len(string) function returns the length of a string.

```
print 'Hello World'[1]

print len('Hello World')
```

Strings

```
s = 'hi'
print s[1]                 ## i
print len(s)               ## 2
print s + ' there'         ## hi there
```

COMMERCE DATA SERVICE

Concatenation

Concatenate strings with **+**. You cannot concatenate different types. The str() function converts values to a string form so they can be combined with other strings.

```
pi = 3.14
text = 'The value of pi is ' + pi       ## NO, does not work
text = 'The value of pi is '  + str(pi) ## yes
```

String Methods

A method is like a function, but it runs "on" an object. If the variable s is a string, then the code s.lower() runs the lower() method on that string object and returns the result (this idea of a method running on an object is one of the basic ideas that make up Object Oriented Programming, OOP).

String Methods

`s.lower(), s.upper()`

returns the lowercase or uppercase version of the string

String Methods

`s.strip()`

returns a string with whitespace removed from the start and end

String Methods

```
s.isalpha()
s.isdigit()
s.isspace()...
```

tests if all the string chars are in the various character classes

String Methods

```
s.startswith('other')
```

```
s.endswith('other')
```

tests if the string starts or ends with the given other string

String Methods

```
s.find('other')
```

searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found

String Methods

```
s.replace('old', 'new')
```

returns a string where all occurrences of 'old' have been replaced by 'new'

String Methods

```
s.split('delim')
```

returns a list of substrings separated by the given delimiter. The delimiter is not a regular expression, it's just text.

String Methods

`'aaa,bbb,ccc'.split(',') -> ['aaa', 'bbb', 'ccc'].`

As a convenient special case s.split() (with no arguments) splits on all whitespace chars.

String Methods

`s.join(list)`

opposite of split(), joins the elements in the given list together using the string as the delimiter.

`'---'.join(['aaa', 'bbb', 'ccc']) -> aaa---bbb---ccc`

String Slices

H	e	l	l	o
0	1	2	3	4
-5	-4	-3	-2	-1

Exercise

Slice the string ‘Star is Here’ to get

‘Here is Star’

String %

The % operator takes a printf-type format string on the left (**%d** int, **%s** string, **%f** or **%g** floating point), and the matching values in a tuple on the right (a tuple is made of values separated by commas, typically grouped inside parentheses)

```
text = "%d is %s" % (3, ‘three’)

text = ("%d little pigs come out or I'll %s and %s and %s" %
    (3, 'huff', 'puff', 'blow down'))
```

Unicode Strings

Regular Python strings are *not* unicode, they are just plain bytes. To create a unicode string, use the 'u' prefix on the string literal:

```
ustring = u'A unicode \u018e string \xf1'
```

Unicode Strings

A unicode string is a different type of object from regular "str" string, but the unicode string is compatible and the various libraries such as regular expressions work correctly if passed a unicode string instead of a regular string.

To convert a unicode string to bytes with an encoding such as 'utf-8', call the ustring.encode('utf-8') method on the unicode string. Going the other direction, the unicode(s, encoding) function converts encoded plain bytes to a unicode string:

```
s = u'A unicode \u018e string \xf1'.encode('utf-8')

t = unicode(s, 'utf-8')
```

If Statements

Python does not use **`{ }`** to enclose blocks of code for if/loops/function etc.. Instead, Python uses the colon (**`:`**) and indentation/whitespace to group statements. The boolean test for an **`if`** does not need to be in parenthesis, and it can have **`elif`** and **`else`** clauses.

Any value can be used as an if-test. The "zero" values all count as false: **`None`**, **`0`**, empty string, empty list, empty dictionary. There is also a Boolean type with two values: **`True`** and **`False`** (converted to an int, these are **`1`** and **`0`**). Python has the usual comparison operations: **`==`**, **`!=`**, **`<`**, **`<=`**, **`>`**, **`>=`**. The boolean operators are the spelled out words **`and`**, **`or`**, **`not`**.

If Statements

```
if speed >= 80:
    print 'License and registration please'
    if mood == 'terrible' or speed >= 100:
        print 'You have the right to remain silent.'
    elif mood == 'bad' or speed >= 90:
        print "I'm going to have to write you a ticket."
        write_ticket()
    else:
        print "Let's try to keep it under 80 ok?"
```

If Statements

```
if speed >= 80: print 'You are so busted'
else: print 'Have a nice day'
```

Mathematical Operators

For numbers, the standard operators, **+**, **-**, **/**, *, **, **%** work in the usual way.

There is no **++** operator, but **+=**, **-=**, etc. work.

If you want integer division, it is most correct to use 2 slashes -- e.g. **6 // 5** is **1**.

Lists

Python has a built-in list type named "list".

List literals are written within square brackets [ ].

Lists work similarly to strings -- use the len() function and square brackets [ ] to access data, with the first element at index 0.

Lists

```
colors = ['red', 'blue', 'green']
print colors[0]    ## red
print colors[2]    ## green
print len(colors)  ## 3
```

Lists

Assignment with an = on lists does not make a copy. Instead, assignment makes the two variables point to the one list in memory.

```
b = colors   ## Does not copy the list
```

Lists

The "empty list" is just an empty pair of brackets [ ].

The '+' works to append two lists, so `[1, 2] + [3, 4]` yields `[1, 2, 3, 4]` (this is just like + with strings).

For Loop

Python's **`for`** and **`in`** constructs are extremely useful, and the first use of them we'll see is with lists.

The **`for`** construct -- **`for var in list`** -- is an easy way to look at each element in a list (or other collection). Do not add or remove from the list during iteration.

For Loop

```
squares = [1, 4, 9, 16]
sum = 0
for num in squares:
   sum += num
print sum  ## 30
```

For Loop

The **`in`** construct on its own is an easy way to test if an element appears in a list (or other collection), returning **`True`**/**`False`**.

```
list = ['larry', 'curly', 'moe']
if 'curly' in list:
   print 'yay
```

For Loop

You can also use **`for`**/**`in`** to work on a string. The string acts like a list of its chars, so **`for ch in s: print ch`** prints all the chars in a string.

Range

The **`range(n)`** function yields the numbers 0, 1, ... n-1, and **`range(a, b)`** returns a, a+1, ... b-1 -- up to but not including the last number. The combination of the for-loop and the **`range()`** function allow you to build a traditional numeric for loop:

```
## print the numbers from 0 through 99
for i in range(100):
   print i
```

While Loop

Python also has the standard while-loop, and the **`break`** and **`continue`**. The above **`for`**/**`in`** loops solves the common case of iterating over every element in a list, but the while loop gives you total control over the index numbers. Here's a while loop which accesses every 3rd element in a list:

```
## Access every 3rd element in a list
i = 0
while i < len(a):
    print a[i]
    i = i + 3
```

List Methods

`list.append(elem)`

adds a single element to the end of the list. Common error: does not return the new list, just modifies the original.

List Methods

`list.insert(index, elem)`

inserts the element at the given index, shifting elements to the right.

List Methods

`list.extend(list2)`

adds the elements in list2 to the end of the list. Using + or += on a list is similar to using extend().

List Methods

`list.index(elem)`

searches for the given element from the start of the list and returns its index. Throws a ValueError if the element does not appear (use "in" to check without a ValueError).

List Methods

`list.remove(elem)`

searches for the first instance of the given element and removes it (throws ValueError if not present)

List Methods

`list.sort()`

sorts the list in place (does not return it). (The sorted() function shown below is preferred.)

List Methods

`list.reverse()`

reverses the list in place (does not return it)

List Methods

`list.pop(index)`

removes and returns the element at the given index. Returns the rightmost element if index is omitted (roughly the opposite of append()).

List Building

One common pattern is to start a list a the empty list [], then use append() or extend() to add elements to it:

```
list = []           ## Start as the empty list
list.append('a')    ## Use append() to add elements
list.append('b')
```

List Slices

Slices work on lists just as with strings, and can also be used to change sub-parts of the list.

```
list = ['a', 'b', 'c', 'd']
print list[1:-1]   ## ['b', 'c']
list[0:2] = 'z'    ## replace ['a', 'b'] with ['z']
print list         ## ['z', 'c', 'd']
```

Sorting

The easiest way to sort is with the **`sorted(list)`** function, which takes a list and returns a new list with those elements in sorted order. The original list is not changed.

```
a = [5, 1, 4, 3]
print sorted(a)  ## [1, 3, 4, 5]
print a          ## [5, 1, 4, 3]
```

Sorting

The **`sorted()`** function can be customized though optional arguments. The **`sorted()`** optional argument **`reverse=True`**, e.g. **`sorted(list, reverse=True)`**, makes it sort backwards.

```
strs = ['aa', 'BB', 'zz', 'CC']
print sorted(strs)  ## ['BB', 'CC', 'aa', 'zz'] (case sensitive)
print sorted(strs, reverse=True)   ## ['zz', 'aa', 'CC', 'BB']
```

Sorting

For more complex custom sorting, sorted() takes an optional "**key=**" specifying a "**key**" function that transforms each element before comparison. The key function takes in 1 value and returns 1 value, and the returned "proxy" value is used for the comparisons within the sort.

For example with a list of strings, specifying **key=len** (the built in **len()** function) sorts the strings by length, from shortest to longest.

```
strs = ['ccc', 'aaaa', 'd', 'bb']
print sorted(strs, key=len)  ## ['d', 'bb', 'ccc', 'aaaa']
```

Sorting

As another example, specifying "str.lower" as the key function is a way to force the sorting to treat uppercase and lowercase the same:

```
## "key" argument specifying str.lower function to use for sorting
print sorted(strs, key=str.lower)  ## ['aa', 'BB', 'CC', 'zz']
```

Sorting

```
strs = ['xc', 'zb', 'yd' ,'wa']

def MyFn(s):
    return s[-1]

print sorted(strs, key=MyFn)  ## ['wa', 'zb', 'xc', 'yd']

print sorted(strs, key=lambda x: x[-1])  ## ['wa', 'zb', 'xc', 'yd']
```

Tuple

A tuple is a fixed size grouping of elements, such as an `(x, y)` co-ordinate.

Tuples are like lists, except they are immutable and do not change size (tuples are not strictly immutable since one of the contained elements could be mutable).

Tuples play a sort of "struct" role in Python. A function that needs to return multiple values can just return a tuple of the values.

Tuple

```
tuple = (1, 2, 'hi')
print len(tuple)  ## 3
print tuple[2]    ## hi
tuple[2] = 'bye'  ## NO, tuples cannot be
changed
tuple = (1, 2, 'bye')  ## this works
```

Tuple

To create a size-1 tuple, the lone element must be followed by a comma.

```
tuple = ('hi',)   ## size-1 tuple
```

Tuple

Assigning a tuple to an identically sized tuple of variable names assigns all the corresponding values. If the tuples are not the same size, it throws an error. This feature works for lists too.

```
(x, y, z) = (42, 13, "hike")
print z  ## hike
```

List Comprehension

A list comprehension is a compact way to write an expression that expands to a whole list. The syntax is **`[ expr for var in list ]`** -- the for var in list looks like a regular for-loop, but without the colon (`:`).

```
nums = [1, 2, 3, 4]
squares = [ n * n for n in nums ]   ## [1, 4, 9, 16]
```

List Comprehension

You can add an if test to the right of the for-loop to narrow the result. The if test is evaluated for each element, including only the elements where the test is true.

```
## Select values <= 2
nums = [2, 8, 1, 6]
small = [ n for n in nums if n <= 2 ]  ## [2, 1]

## Select fruits containing 'a', change to uppercase
fruits = ['apple', 'cherry', 'banana', 'lemon']
afruits = [ s.upper() for s in fruits if 'a' in s ]
## ['APPLE', 'BANANA']
```

Exercise

Given `a = range(100)` square every odd number using list comprehension

Dictionary

Python's efficient key/value hash table structure is called a "dict".

The contents of a dict can be written as a series of key:value pairs within braces `{}`, e.g. **`dict = {key1:value1, key2:value2, ... }`**. The "empty dict" is just an empty pair of curly braces `{}`.

Looking up or setting a value in a dict uses square brackets, e.g. **`dict['foo']`** looks up the value under the key **`'foo'`**. Strings, numbers, and tuples work as keys, and any type can be a value.

Dictionary

```
dict = {}
dict['a'] = 'alpha'
dict['g'] = 'gamma'
dict['o'] = 'omega'
```

Dictionary

```
print dict  ## {'a': 'alpha', 'o': 'omega', 'g': 'gamma'}
```

Dictionary

```
print dict['a']     ## Simple lookup, returns 'alpha'
dict['a'] = 6       ## Put new key/value into dict
'a' in dict         ## True

## print dict['z']                  ## Throws KeyError

if 'z' in dict: print dict['z']     ## Avoid KeyError
print dict.get('z')  ## None (instead of KeyError)
```

Dictionary

A for loop on a dictionary iterates over its keys by default.

The keys will appear in an arbitrary order.

```
for key in dict: print key
```

Dictionary

The methods **`dict.keys()`** and **`dict.values()`** return lists of the keys or values explicitly.

```
for key in dict.keys(): print key

print dict.values()
```

Dictionary

There's also an **`dict.items()`** which returns a list of **`(key, value)`** tuples, which is the most efficient way to examine all the key value data in the dictionary. All of these lists can be passed to the **`sorted()`** function.

```
print dict.items()

for key in sorted(dict.keys()):
   print key, dict[key]
```

Dictionary Formatting

The % operator works conveniently to substitute values from a dict into a string by name:

```
hash = {}
hash['word'] = 'garfield'
hash['count'] = 42
s = 'I want %(count)d copies of %(word)s' % hash
```

Del

The "del" operator does deletions. In the simplest case, it can remove the definition of a variable, as if that variable had not been defined. Del can also be used on list elements or slices to delete that part of the list and to delete entries from a dictionary.

```
var = 6
del var  # var no more!
```

Input/Output

The **`open()`** function opens and returns a file handle that can be used to read or write a file in the usual way.

The code **`f = open('name', 'r')`** opens the file into the variable **`f`**, ready for reading operations, and use **`f.close()`** when finished.

Input/Output

Instead of `'r'`, use `'w'` for writing, and `'a'` for append.

Input/Output

The special mode `'rU'` is the "Universal" option for text files where it's smart about converting different line-endings so they always come through as a simple `'\n'`.

Input/Output

```
f = open('foo.txt', 'rU')
for line in f:
  print line,
f.close()
```

Input/Output

```
with open('foo.txt', 'rU') as f:
  for line in f:
    print line,
```

Import

We can import additional packages to extend Python:

```
import ujson as json

import json
```

JSON/uJSON

JSON is an open-standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs.

JSON/uJSON

json is a default package that is included with Python. It allows for the decoding and encoding of JSON format.

UltraJSON or ujson is an additional package that allows for faster decoding and encoding of JSON format.

JSON/uJSON

```
{
  "firstName": "John",
  "lastName": "Smith",
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

JSON/uJSON

```
import json
with open('file.json', 'rU') as f:
  loadedfile = json.load(f)
```

JSON/uJSON

```
import json
assembledfile = {‘a’:’1’,’b’:’2’}
with open(‘file.json’, ‘wU’) as f:
  json.dump(assembledfile,f)
```

NumPy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- **a powerful N-dimensional array object**
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- **useful linear algebra, Fourier transform, and random number capabilities**

NumPy

```
import numpy as np

a = range(10)

print ‘DescriptiveStatistics! %f, %f’%(np.mean(a),np.std(a))
```