Lesson 04: Container Data Types

0

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

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

3 2 838 415

python fcs6

(U) Lesson 04: Container Data Types

Recommendations

UNCLASSIFIED

(U) Introduction

(U) Now that we've worked with strings and numbers, we turn our attention to the next logical thing: data containers that allow us to build up complicated structures. There are different ways of putting data into containers, depending on what we need to do with it, and Python has several built-in containers to support the most common use cases. Python's built-in container types include:

1. `list`
2. `tuple`
3. `dict`
4. `set`
5. `frozenset`

(U) Of these, `tuple` and `frozenset` are **immutable**, which means that they can not be changed after they are created, whether that's by addition, removal, or some other means. Numbers and strings are also immutable, which should make the following statement more sensible:
the **variable** that names an immutable object can be reassigned, but the immutable object itself can't be changed.

(U) To create an instance of any container, we call its name as a function (sometimes known as a *constructor*). With no arguments, we get an empty instance, which isn't very useful for immutable types. Shortcuts for creating non-empty `list` s, `tuple` s, `dict` s, and even `set` s will be covered in the following sections.

```
list()
```

```
dict()
```

```
tuple()
```

```
set()
```

(U) Many built-in functions and even some operators work with container types, where it makes sense. Later on we'll see the behind-the-scenes mechanism that makes this work; for now, we'll enumerate how this works as part of the discussion of each separate type.

(U) Lists

(U) A `list` is an ordered sequence of zero or more objects, which are often of different types. It is commonly created by putting square brackets `[ ]` around a comma-separated list of its initial values:

```
a = ['spam', 'eggs', 5, 3.2, [100, 200, 300]]
```

```
fruit = ['Apple', 'Orange', 'Pear', 'Lime']
```

(U) Values can be added to or removed from the list in different ways:

```
fruit.append('Banana')
```

```
fruit.insert(3, 'Cherry')
```

```
fruit.append(['Kiwi', 'Watermelon'])
```

```
fruit.extend(['Cherry', 'Banana'])
```

```
fruit.remove('Banana')
```

```
fruit
```

```
fruit.pop()
```

```
fruit.pop(3)
```

```
fruit
```

(U) The `+` operator works like the `extend` method, except that it returns a **new list**.

```
a + fruit
```

```
a
```

```
fruit
```

(U) Other operators and methods tell how long a list is, whether an element is in the list, and if so, where or how often it is found.

```
len(fruit)
```

```
fruit.append('Apple')
```

```
'Apple' in fruit
```

```
'Cranberry' not in fruit
```

```
fruit.count('Apple')
```

```
fruit.index('Apple')     # Careful--can cause an error
```

```
fruit.index('Apple', 1)
```

(U) List Comprehension

(U) Great effort has been to make lists easy to work with. One of the most common uses of a list is to iterate over its elements with a `for` loop, storing off the results of each iteration in a new list. Python removes the repetitive boilerplate code from this type of procedure with **list comprehensions**. They're best learned by example:

```
a = [i for i in range(10)]
```

```
b = [i**2 for i in range(10)]
```

```
c = [[i, i**2, i**3] for i in range(10)]
```

```
d = [[i, i**2, i**3] for i in range(10) if i % 2]  # conditionals!
```

```
e = [[i+j for i in 'abcde'] for j in 'xyz']      # nesting!
```

(U) Sorting and Reordering

(U) Sorting is another extremely common operation on lists. We'll cover it in greater detail later, but here we cover the most basic built-in ways of sorting. The `sorted` function works on more than just `list` s, but always returns a new list with the same contents as the original in sorted order. There is also a `sort` method on `list` s that performs an in-place sort.

```
fruit.remove(['Kiwi', 'Watermelon'])  # can't compare list with str
sorted_fruit = sorted(fruit)
```

```
sorted_fruit == fruit
```

```
fruit.sort()
```

```
sorted_fruit == fruit
```

(U) Reversing the order of a list is similar, with a built-in `reversed` function and an in-place `reverse` method for `list` s. The `reversed` function returns an iterator, which must be converted back into a list explicitly. To sort something in reverse, you *could* combine the `reversed` and the `sorted` methods, but you *should* use the optional `reverse` argument on the `sorted` and `sort` functions.

```
r_fruit = list(reversed(fruit))
```

```
fruit.reverse()
```

```
r_fruit == fruit
```

```
sorted(r_fruit, reverse=True)
```

(U) Tuples

(U) Much like a `list` , a `tuple` is an ordered sequence of zero or more objects of any type. They can be constructed by putting a comma-separated list of items inside parentheses `( )` , or even by assigning a comma-separated list to a variable with no delimiters at all. Parentheses are heavily overloaded--they also indicate function calls and mathematical order of operations--so defining a one-element tuple is tricky: the one element must be followed by a comma. Because a `tuple` is **immutable**, it won't have any of the methods that change lists, like `append` or `sort` .

```
a = (1, 2, 'first and second')
```

```
len(a)
```

```
sorted(a)
```

```
a.index(2)
```

```
a.count(2)
```

```
b = '1', '2', '3'
```

```
type(b)
```

```
c_raw = '1'
```

```
c_tuple = '1',
```

```
c_raw == c_tuple
```

```
d_raw = ('d')
```

```
d_tuple = ('d',)
```

```
d_raw == d_tuple
```

(U) Interlude: Index and Slice Notation

(U) For the ordered containers `list` and `tuple`, as well as for other ordered types like `str`ings, it's often useful to retrieve or change just one element or a subset of the elements. *Index*and *slice* notation are available to help with this. Indexes in Python always start at 0. We'll start out with a new list and work by example:

```
animals = ['tiger', 'monkey', 'cat', 'dog', 'horse', 'elephant']
```

```
animals[1]
```

```
animals[1] = 'chimpanzee'
```

```
animals[1:3]
```

```
animals[3] in animals[1:3]
```

```
animals[:3]       # starts at beginning
```

```
animals[4:]       # goes to the end
```

```
animals[-2:]
```

```
animals[1:6:2]    # uses the optional step parameter
```

```
animals[::-1] == list(reversed(animals))
```

(U) Because slicing returns a new list and not just a view on the list, it can be used to make a copy (technically a `shallow` copy):

```
same_animals = animals
```

```
different_animals = animals[:]
```

```
same_animals[0] = 'lion'
```

```
animals[0]
```

```
different_animals[0] = 'leopard'
```

```
different_animals[0] == animals[0]
```

(U) Dictionaries

(U) A `dict` is a container that associates keys with values. The keys of a `dict` must be unique, and only immutable objects can be keys. Values can be any type.

(U) The dictionary construction shortcut uses curly braces `{ }` with a colon `:` between keys and values (e.g. `my_dict = {key: value, key1: value1}`). Alternate constructors are available using the `dict` keyword. Values can be added, changed, or retrieved using index notation with *keys* instead of *index numbers*. Some of the operators, functions, and methods that work on sequences also work with dictionaries.

```
bugs = {"ant": 10, "praying mantis": 0}
```

```
bugs['fly'] = 5
```

```
bugs.update({'spider': 1})     # like extend
```

```
del bugs['spider']
```

```
'fly' in bugs
```

```
5 in bugs
```

```
bugs['fly']
```

(U) Dictionaries have several additional methods specific to their structure. Methods that return lists, like `items` , `keys` , and `values` , are not guaranteed to do so in any particular order, but may be in consistent order if no modifications are made to the dictionary in between the calls. The `get` method is often preferable to index notation because it does not raise an error when the requested key is not found; instead, it returns `None` by default, or a default value that is passed as a second argument.

```
bugs.items() # list of tuples
```

```
bugs.keys()
```

```
bugs.values()
```

```
bugs.get('fly')
```

```
bugs.get('spider')
```

```
bugs.get('spider', 4)
```

```
bugs.clear()
```

```
bugs
```

(U) Sets and Frozensets

(U) A `set` is a container that can only hold unique objects. Adding something that's already there will do nothing (but cause no error). Elements of a set must be immutable (like keys in a dictionary). The `set` and `frozenset` constructors take any iterable as an argument, whether it's a `list`, `tuple`, or otherwise. Curly braces `{ }` around a list of comma-separated values can be used in Python 2.7 and later as a shortcut constructor, but that could cause confusion with the `dict` shortcut. Two sets are equal if they contain the same items, regardless of order.

```
numbers = set([1,1,1,1,1,3,3,3,3,3,2,2,2,3,3,4])
```

```
letters = set('TheQuickBrownFoxJumpedOverTheLazyDog'.lower())
```

```
a = {} # dict
```

```
more_numbers = {1, 2, 3, 4, 5} # set
```

```
numbers.add(4)
```

```
numbers.add(5)
```

```
numbers.update([3, 4, 7])
```

```
numbers.pop()              # could be anything
```

```
numbers.remove(7)
```

```
numbers.discard(7)          # no errror
```

(U) A frozen set is constructed in a similar way; the only difference is in the mutability. This makes frozen sets suitable as dictionary keys, but frozen sets are uncommon.

```
a = frozenset([1,1,1,1,1,3,3,3,3,32,2,2,3,3,4])
```

(U) Sets adopt the notation of bitwise operators for set operations like *union*, *intersection*, and *symmetric difference*. This is similar to how the `+` operator is used for concatenating `list` s and `tuple` s.

```
house_pets = {'dog', 'cat', 'fish'}
```

```
farm_animals = {'cow', 'sheep', 'pig', 'dog', 'cat'}
```

```
house_pets & farm_animals   # intersection
```

```
house_pets | farm_animals   # union
```

```
house_pets ^ farm_animals   # symmetric difference
```

```
house_pets - farm_animals   # asymmetric difference
```

(U) There are verbose set methods that do the same thing, but with two important difference: they accept `list` s, `tuple` s, and other iterables as arugments, and can be used to update the set *in place*. Although there are methods corresponding to all the set operators, we give only a few examples.

```
farm_animal_list = list(farm_animals) * 2
```

```
house_pets.intersection(farm_animal_list)
```

```
house_pets.union(farm_animal_list)
```

```
house_pets.intersection_update(farm_animal_list)
```

(U) Comparison of sets is similar: operators can be used to compare two sets, while methods can be used to compare sets with other iterables. Unlike numbers or strings, sets are often incomparable.

```
house_pets = {'dog', 'cat', 'fish'}
```

```
farm_animals > house_pets
```

```
house_pets < farm_animals
```

```
house_pets.intersection_update(farm_animals)
```

```
farm_animals > house_pets
```

```
house_pets.issubset(farm_animal_list)
```

(U) Coda: More Built-In Functions

(U) We've seen how some built-in functions operate on one or two of these container types, but all of the following can be applied to any container, although they probably won't always work; that depends on the contents of the container. There are some caveats:

- (U) When passed a dictionary as an argument, these functions look at the keys of the dictionary, not the values.
- (U) The `any` and `all` functions use the boolean context of the values of the container, e.g. `0` is `False` and non-zero numbers are `True`, and all strings are `True` except for the empty string `''`, which is `False`.
- (U) The `sum` function only works when the contents of the container are numbers.

```
generic_container = farm_animals    # or bugs, animals, etc.
```

```
all(generic_container)
```

```
any(generic_container)
```

```
'pig' in generic_container
```

```
'pig' not in generic_container
```

```
len(generic_container)
```

```
max(generic_container)
```

```
min(generic_container)
```

```
sum([1, 2, 3, 4, 5])
```

Lesson 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.

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?

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'

UNCLASSIFIED

Lesson 05: File Input and Output

0

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

3 938 414

python fcs6

(U) Lesson 05: File Input and Output

Recommendations

UNCLASSIFIED

(U) Introduction: Getting Dangerous

(U) As you probably already know, input and output is a core tool in algorithm development and reading from and writing to files is one of the most common forms. Let's jump right in just to see how easy it is to write a file.

```
myfile = open('data.txt', 'w')
myfile.write("I am writing data to my file")
myfile.close()
```

(U) And there you have it! You can write data to files in Python. By the way, the variables you put into that `open` command are the filename (as a string--do not forget the path) and the file *mode*. Here we are writing the file, as indicated by the `'w'` as the second argument to the `open` function.

(U) Let tear apart what we actually did.

```
open('data.txt', 'w')
```

(U) This actually returns something called a *file object*. Let's name it!

(U) Danger: Opening a file that already exists for writing **will erase the original file**.

```
myfile = open('data.txt', 'w')
```

(U) Now we have a variable to this file object, which was opened in write mode. Let's try to write to the file:

```
myfile.write("I am writing data to my file")
```

```
myfile.read()   # Oops...notice the error
```

```
myfile.close()  # Guess what that did...
```

(U) There are only a few file modes which we need to use. You have seen `'w'` (writing). The others are `'r'` (reading), `'a'` (appending), `'r+'` (reading and writing), and `'b'` (binary mode).

```
myfile = open('data.txt', 'r')
```

```
myfile.read()
```

```
myfile.write("I am writing more data to my file")  # Oops again...check our mode
```

```
mydata = myfile.read()
```

```
mydata          # HEY! Where did the data go....
```

```
myfile.close()  # don't be a piggy
```

(U) A cool way to use contents of a file in a block is with the `with` command. Formally, this is called a *context manager*. Informally, it ensures that the file is closed when the block ends.

```
with open('data.txt') as f:
    print(f.read())
```

(U) Using `with` is a good idea but is usually not absolutely necessary. Python tries to close files once they are no longer needed. Having files open is not usually a problem, unless you try to open a large number all at once (e.g. inside a loop).

(U) Reading Lines From Files

(U) Here are some of the other useful methods for file objects:

```
lines_file = open('fewlines.txt', 'w')
```

```
lines_file.writelines("first\n")
```

```
lines_file.writelines(["second\n", "third\n"])
```

```
lines_file.close()
```

(U) Similarly:

```
lines_file = open('fewlines.txt', 'r')
```

```
lines_file.readline()
```

```
lines_file.readline()
```

```
lines_file.readline()
```

```
lines_file.readline()
```

(U) And make sure the file is closed before opening it up again in the next cell

```
lines_file.close()
```

(U) Alternately:

```
lines = open('fewlines.txt', 'r').readlines()  # Note the plurality
```

```
lines
```

(U) Note: both `read` and `readline(s)` have optional size arguments that limit how much is read. For `readline(s)`, this may return incomplete lines.

(U) But what if the file is very long and I don't need or want to read all of them at once. `file` objects behave as their own iterator.

```
lines_file = open('fewlines.txt', 'r')
```

```
for line in lines_file:
    print(line)
```

The below syntax is a very common formula for reading through files. Use the `with` keyword to make sure everything goes smoothly. Loop through the file one line at a time, because often our files have one record to a line. And do something with each line.

```
with open('fewlines.txt') as my_file:
    for line in my_file:
        print(line.strip()) # The strip function removes newlines and whitespace from the start and finish
```

The file was closed upon exiting the `with` block.

(U) Moving Around With `tell` and `seek`

(U) The `tell` method returns the current position of the cursor within the file. The `seek` command sets the current position of the cursor within the file.

```
inputfile = open('data.txt', 'r')
```

```
inputfile.tell()
```

```
inputfile.read(4)
```

```
inputfile.tell()
```

```
inputfile.seek(0)
```

```
inputfile.read()
```

(U) File-Like objects

(U) There are other times when you really need to have data in a file (because another function requires it be read from a file perhaps). But why waste time and disk space if you already have the data in memory?

(U) A very useful module to make a string into a file-like object is called `StringIO`. This will take a string and give it file methods like `read` and `write`.

```
import io
```

```
mystringfile = io.StringIO()            # For handing bytes, use io.BytesIO
```

```
mystringfile.write("This is my data!")  # We just wrote to the object, not a filehandle
```

```
mystringfile.read()                     # Cursor is at the end!
```

```
mystringfile.seek(0)
```

```
mystringfile.read()
```

```
newstringfile = io.StringIO("My data")  # The cursor will automatically be set to 0
```

(U) Now let's pretend we have a function that expects to read data from a file before it operates on it. This sometimes happens when using library functions.

```
def iprintdata(f):
    print(f.read())
```

```
iprintdata('mydata')      # Grrr!
my_io = io.StringIO('mydata')
iprintdata(my_io)         # YAY!
```

Lesson Exercises

Get the data

Copy sonnet from https://urn.nsa.ic.gov/t/tx6qm and paste into 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.

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

Exercise 2

Write 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': 2, ... }
```

Extra Credit

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

Lesson 06: Development Environment and Tooling

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

1

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

3 1 407 96

fcs6 extra interactive numpy python requests

(U) Lesson 06: Development Environment and Tooling

Recommendations

(U) Package Management

(U) **The Problem:** Python has a "batteries included" philosophy–it has a comprehensive standard library, but by default, using other packages leaves something to be desired:

- Python doesn't have a `classpath`, and unless you are `root`, you can't install new packages for the whole system.
- How do you share a script with someone else when you don't know what packages are installed on their system?
- Sometimes you have to use **Project A**, which relies on a package that requires **awesome-package v.1.1**, but you're writing **Project B** and want to use some features that are new in **awesome-package v.2.0**?
- The best-in-class package manager isn't in the Python standard library.

(U) The Solution: `virtualenv`

(U) The `virtualenv` package creates **virtual environments**, i.e. isolated spaces containing their own Python instances. It provides a utility script that manipulates your environment to **activate** your environment of choice.

(U) It's already installed and available on the class VM. The `-p` flag indicates which Python executable to use as the base for the virtual environment:

```
[redacted] ~]$ virtualenv NEWENV -p /usr/local/bin/python
New python executable in NEWENV/bin/python
Installing Setuptools.....................................................done.
Installing Pip............................................................done.
[redacted] ~]$ which python
/usr/local/bin/python
[redacted] ~]$ source NEWENV/bin/activate
(NEWENV)[redacted] ~]$ which python
~/NEWENV/bin/python
(NEWENV)[redacted] ~]$ deactivate
[redacted] ~]$
```

P.L. 86-36

(U) The `virtualenv` package can be downloaded and run as a script to create a virtual environment based on any recent Python installation. A virtual environment has the package manager `pip` pre-installed, which can be hooked into the internal mirror of the Python Package Index (PyPI) by exporting the correct address to the `PIP_INDEX_URL` environment variable:

```
[redacted] ~]$ echo $PIP_INDEX_URL
http://bbtux022.gp.proj.nsa.ic.gov/PYPI
[redacted] ~]$ python
Python 2.7.5 (default, Nov  6 2013, 10:23:48)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

```
import requests
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named requests
```

```
exit()
```

Doc ID: 6689693

```
[redacted] ~]$ source NEWENV/bin/activate
[redacted] ~]$ source NEWENV/bin/activate
(NEWENV)[redacted] ~]$ pip install requests
Downloading/unpacking requests
  Downloading requests-2.0.0.tar.gz (362kB): 362kB downloaded
  Running setup.py egg_info for package requests

Installing collected packages: requests
  Running setup.py install for requests

Successfully installed requests
Cleaning up...
(NEWENV)[redacted] ~]$ python
Python 2.7.5 (default, Nov  6 2013, 10:23:48)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

```
import requests
```

```
requests.__version__
```

```
'2.0.0'
```

```
import sys
```

```
sys.path
```

```
['', '/home/[redacted]/NEWENV/lib/python27.zip', '/home/[redacted]/NEWENV/lib/python2.7', '/home/[redacted]/NEWENV/lib/python2.7/plat-
linux2', '/home/[redacted]/NEWENV/lib/python2.7/lib-tk', '/home/[redacted]/NEWENV/lib/python2.7/lib-old', '/home/[redacted]/NEWENV/li
b/python2.7/lib-dynload', '/usr/local/lib/python2.7', '/usr/local/lib/python2.7/plat-linux2', '/usr/local/lib/python2.7/lib-t
k', '/home/[redacted]/NEWENV/lib/python2.7/site-packages']
```

```
exit()
```

```
(NEWENV)[redacted] ~]$ pip freeze
requests==2.0.0
wsgiref==0.1.2
```

P.L. 86-36

Now we have a place to install custom code and a way to share it!

- Develop code inside `~/NEWENV/lib/python2.7/site-packages`
- Capture installed packages with `pip freeze >> requirements.txt` and install them to a new `virtualenv` with `pip install -r requirements.txt` .

(U) The Ultimate Package

(U) `IPython` is an alternative interactive shell for Python with lots of cool features, among which are:

- tab completion,
- color output,
- rich history recall,
- better help interface,
- 'magic' commands,
- a web-based notebook interface with easy-to-share files, and
- distributed computing (don't ask about this)

(U) To get started:

Doc ID: 6689693

```
(NEWENV)[                ~]$ pip install ipython
  Downloading ipython-1.1.0.tar.gz (8.7MB): 8.7MB downloaded
...
Successfully installed ipython
Cleaning up...
(NEWENV)[                ~]$ ipython
Python 2.7.5 (default, Nov  6 2013, 10:23:48)
Type "copyright", "credits" or "license" for more information.

IPython 1.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: ls
BASE3/  Hello World.html  Hello World.ipynb  NEWENV/

In[2]: hist
ls
hist

In[3]: import os

In[4]: os.path #press tab
os.path            os.pathconf        os.pathconf_names  os.pathsep

In [4]: os.path
```

P.L. 86-36

(U) To use the web interface, you have to install supplemental packages:

```
(NEWENV)[                ~]$ pip install pyzmq tornado jinja2 pygments
(NEWENV)[                ~]$ ipython notebook --no-mathjax
```

(U) Just two more packages are required to get awesome inline graphics

```
(NEWENV)[                ~]$ pip install numpy
(NEWENV)[                ~]$ pip install matplotlib
```

Lesson 07: Object Orienteering: Using Classes

0

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

Updated 9 months ago by [redacted] in COMP 3321

3 4 721 356

python fcs6

(U) Introduction to classes, objects, and inheritance in Python.

Recommendations

UNCLASSIFIED

(U) Introduction

(U) From the name of it you can see that **object-oriented programming** is oozing with abstraction and complication. Take heart: there's no need to fear or avoid object-oriented programming in Python! It's just another easy-to-use, flexible, and dynamic tool in the deep toolbox that Python makes available. In fact, we've been *using* objects and object oriented concepts ever since the first line of Python code that we wrote, so it's already familiar. In this lesson, we'll think more deeply about what it is that we've been doing all along, and how we can take advantage of these ideas.

(U) Consider, for example, the difference between a **function** and a **method**:

```
name = "Mark"

len(name)     # function

name.upper()  # method
```

(U) In this example, `name` is an **instance** of the `str` **type**. In other words, `name` is an **object** of that type. An **object** is just a convenient wrapper around a combination of some *data* and *functionality* related to that data, embodied in **methods**. Until now, you've probably thought of every `str` just in terms of its data, i.e. the literal string `"Mark"` that was used to assign the variable. The **methods** that work with `name` were defined just once, in a **class definition**, and apply to every string that is ever created. **Methods** are actually the same thing as functions that live *inside* a class instead of *outside* it. (This paragraph probably still seems really confusing. Try re-reading it at the end of the lesson!)

(U) Your First `class`

(U) Just as the keyword `def` is used to define functions, the keyword `class` is used to define a `type` object that will generate a new kind of object, which you get to name!. As an ongoing example, we'll work with a class that we'll choose to name `Person`:

```
class Person(object):
    pass
```

```
type(Person)
```

```
type(Person) == type(int)
```

```
nobody = Person()
```

```
type(nobody)
```

(U) At first, the `Person` class doesn't do much, because it's totally empty! This isn't as useless as it seems, because, just like everything else in Python, classes and their objects are *dynamic*. The `(object)` after `Person` is not a function call; here it names the parent class. Even though the `Person` class looks boring, the fundamentals are there:

- the `Person` class is just as much of a class as `int` or any other built-in,
- we can make an *instance* by using the class name as a constructor function, and
- the `type` of the instance `nobody` is `Person`, just like `type(1)` is `int`.

(U) Since that's about all we can do, let's start over, and wrap some data and functionality into the `Person`:

```
class Person(object):
    species = "Homo sapiens"
    def talk(self):
        return "Hello there, how are you?"
```

```
nobody = Person()
```

```
nobody.species
```

```
nobody.talk()
```

(U) It's **very important** to give any method (i.e. function defined in the class) at least one argument, which is almost always called `self`. This is because internally Python translates `nobody.talk()` into something like `Person.talk(nobody)`.

(U) Let's experiment with the `Person` class and its objects and do things like re-assigning other data attributes.

```
somebody = Person()
```

Doc ID: 6689693

```
somebody.species = 'Homo internetus'
```

```
somebody.name = "Mark"
```

```
nobody.species
```

```
Person.species = "Unknown"
```

```
nobody.species
```

```
somebody.species
```

```
Person.name = "Unknown"
```

```
nobody.name
```

```
somebody.name
```

```
del somebody.name
```

```
somebody.name
```

(U) Although we could add a `name` to each instance just after creating it, one at a time, wouldn't it be nice to assign instance-specific attributes like that when the object is first constructed? The `__init__` function lets us do that. Except for the funny underscores in the name, it's just an ordinary function; we can even give it default arguments.

```
class Person(object):
    species = "Homo sapiens"
    def __init__(self, name="Unknown", age=18):
        self.name = name
        self.age = age
    def talk(self):
        return "Hello, my name is {}.".format(self.name)
```

```
mark = Person("Mark", 33)
```

```
generic_voter = Person()
```

```
generic_worker = Person(age=41)
```

```
generic_worker.age
```

```
generic_worker.name
```

(U) In Python, it isn't unusual to access attributes of an object directly, unlike some languages (e.g. Java), where that is considered poor form and everything is done through getter and setter methods. This is because in Python, attributes can be added and removed at any time, so the getters and setters might be useless by the time that you want to use them.

```
mark.favorite_color = "green"
```

```
del generic_worker.name
```

```
generic_worker.name
```

(U) One potential downside is that Python has no real equivalent of *private* data and methods; everyone can see everything. There is a polite *convention*: other developers are *supposed* to treat an attribute as private if its name starts with a single underscore (`_`). And there is also a *trick*: names that start with two underscores (`__`) are mangled to make them harder to access.

(U) The `__init__` method is just one of many that can help your `class` behave like a full-fledged built-in Python object. To control how your object is printed, implement `__str__` , and to control how it looks as an output from the interactive interpreter, implement `__repr__` . This time, we won't start from scratch; we'll add these dynamically.

```
def person_str(self):
    return "Name: {0}, Age: {1}".format(self.name, self.age)
```

```
Person.__str__ = person_str
```

```
def person_repr(self):
    return "Person('{0}',{1})".format(self.name, self.age)
```

```
Person.__repr__ = person_repr
```

```
print(mark)  # which special method does print use?
```

```
mark         # which special method does Jupyter use to auto-print?
```

(U) Take a minute to think about what just happened:

- We added methods to a class after making a bunch of objects, but *every object* in that class was immediately able to use that method.
- Because they were *special methods*, we could immediately use built-in Python functions (like `str`) on those objects.

(U) Be careful when implementing special methods. For instance, you might want the default sort of the `Person` class to be based on age. The special method `__lt__(self,other)` will be used by Python in place of the built-in `lt` function, even for sorting. (Python 2 uses `__cmp__` instead.) Even though it's easy, this is problematic because it makes objects appear to be equal when they are just of the same age!

```
def person_eq(self, other):
    return self.age == other.age
```

```
Person.__eq__ = person_eq
```

```
bob = Person("Bob", 33)
```

```
bob == mark
```

(U) In a situation like this, it might be better to implement a subset of the **rich comparison** methods, maybe just `__lt__` and `__gt__`, or use a more complicated `__eq__` function that is capable of uniquely identifying all the objects you will ever create.

(U) While we've shown examples of adding methods to a class after the fact, note that it is rarely actually done that way in practice. Here we did that just for convenience of not having to re-define the class every time we wanted to create a new method. Normally you would just define all class methods under the class itself. If we were to do so with the `__str__`, `__repr__`, and `__eq__` methods for the `Person` class above, the class would like the below:

```
class Person(object):
    species = "Homo sapiens"
    def __init__(self, name="Unknown", age=18):
        self.name = name
        self.age = age
    def talk(self):
        return "Hello, my name is {}.".format(self.name)
    def __str__(self):
        return "Name: {0}, Age: {1}".format(self.name, self.age)
    def __repr__(self):
        return "Person('{0}',{1})".format(self.name, self.age)
    def __eq__(self, other):
        return self.age == other.age
```

(U) Inheritance

(U) There are many types of people, and each type could be represented by its own class. It would be a pain if we had to reimplement the fundamental `Person` traits in each new class. Thankfully, **inheritance** gives us a way to avoid that. We've already seen how it works: `Person` inherits from (or is a **subclass** of) the `object` class. However, any class can be inherited from (i.e. have *descendants*).

```
class Student(Person):
    bedtime = 'Midnight'
    def do_homework(self):
        import time
        print("I need to work.")
        time.sleep(5)
        print("Did I just fall asleep?")
```

```
tyler = Student("Tyler", 19)
```

```
tyler.species
```

```
tyler.talk()
```

```
tyler.do_homework()
```

(U) An object from the subclass has all the properties of the parent class, along with any additions from its own class definition. You can still easy to override behavior from the parent class easily--just create a method with the same name in the subclass. Using the parent class's behavior in the child class is tricky, but fun, because you have to use the `super` function.

```
class Employee(Person):
    def talk(self):
        talk_str = super(Employee, self).talk()
        return talk_str + "  I work for {}".format(self.employer)
```

```
fred = Employee("Fred Flintstone", 55)
```

```
fred.employer = "Slate Rock and Gravel Company"
```

```
fred.talk()
```

(U) The syntax here is strange at first. The `super` function takes a `class` (i.e. a `type`) as its first argument, and an object descended from that class as its second argument. The object has a chain of ancestor classes. For `fred`, that chain is `[Employee, Person, object]`.
The `super` function goes through that chain and returns the class that is *after* the one passed as the function's first argument.
Therefore, `super` can be used to skip up the chain, passing modifications made in intermediate classes.

(U) As a second, more common (but more complicated) example, it's often useful to add additional properties to subclass objects in the constructor.

```
class Employee(Person):
    def __init__(self, name, age, employer):
        super(Employee, self).__init__(name, age)
        self.employer = employer
    def talk(self):
        talk_str = super(Employee, self).talk()
        return talk_str + "  I work for {}".format(self.employer)
```

```
fred = Employee("Fred Flintstone", 55, "Slate Rock and Gravel Company")
```

```
fred.talk()
```

(U) A `class` in Python can have more than one listed ancestor (which is sometimes called *polymorphism*). We won't go into great detail here, aside from pointing out that it exists and is powerful but complicated.

```
class StudentEmployee(Student, Employee):
    pass
```

```
ann = StudentEmployee("ann", 58, "Family Services")
```

```
ann.talk()
```

```
bill = StudentEmployee("bill", 20)  # what happens here? why?
```

(U) Lesson Exercises

(U) Exercise 1

(U) Write a Query class that has the following attributes:

- classification
- justification
- selector

(U) Provide default values for each attribute (consider using `None`). Make it so that when you print it, you can display all of the attributes and their values nicely.

```
# your class definition here
```

(U) Afterwards, something like this should work:

```
query1 = Query("TS//SI//REL TO USA, FVEY", "Primary email address of Zendian diplomat", "ileona@stato.gov.zd")
```

```
print(query1)
```

(U) Exercise 2

(U) Make a RangedQuery class that inherits from Query and has the additional attributes:

- begin date
- end date

(U) For now, just make the dates of the form *YYYY-MM-DD*. Don't worry about date formatting or error checking for now. We'll talk about the `datetime` module and exception handling later.

(U) Provide defaults for these attributes. Make sure you incorporate the Query class's initializer into the RangedQuery initializer. Ensure the new class can also be printed nicely.

```
# your class definition here
```

(U) Afterwards, this should work:

```
query2 = RangedQuery("TS//SI//REL TO USA, FVEY", "Primary IP address of Zendian diplomat", "10.254.18.162", "2016-12-01", "201
```

```
print(query2)
```

(U) Exercise 3

(U) Change the Query class to accept a list of selectors rather than a single selector. Make sure you can still print everything OK.

UNCLASSIFIED

Lesson 07: Supplement

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

0

Updated 11 months ago by [redacted] in COMP 3321

3 84 40

fcx91

(U) Supplement to lesson 07 based on exercises from previous lectures.

Recommendations

You may have written a function like this to check if an item is in your grocery list and print something snarky if it's not:

```
def in_my_list(item):
    my_list = ['apples','milk','butter','orange juice']
    if item in my_list:
        return 'Got it!'
    else:
        return 'Nope!'
```

```
in_my_list('apples')
```

```
in_my_list('chocolate')
```

But what if I really wanted chocolate to be on my list? I would have to rewrite my function. If I had written a class instead of a function, I would be able to change my list.

```
class My_list(object):
    my_list = ['apples','milk','butter','orange juice']
    def in_my_list(self,item):
        if item in self.my_list:
            return 'Got it!'
        else:
            return 'Nope!'
```

```
december = My_list()
```

```
december.in_my_list('chocolate')
```

```
december.my_list =december.my_list +['chocolate']
```

```
december.in_my_list('chocolate')
```

Now I have a nice template for grocery lists and grocery list behavior

```
jan = My_list()
```

```
december.my_list
```

```
jan.my_list
```

This isn't helpful:

```
print(december)
```

So we overwrite the __str__ function we inherited from object:

```
class My_list(object):
    my_list = ['apples','milk','butter','orange juice']

    def __str__(self):
        return 'My list: {}'.format(', '.join(self.my_list))
    def __repr__(self):
        return self.__str__()
    def in_my_list(self,item):
        if item in self.my_list:
            return 'Got it!'
        else:
            return 'Nope!'
```

```
december = My_list()
print(december)
```

```
december
```

Maybe I also want to be more easily test if my favorite snack is on the list...

Doc ID: 6689693

```
class My_list(object):
    my_list = ['apples','milk','butter','orange juice']
    def __init__(self,snack='chocolate'):
        self.snack = snack
    def __str__(self):
        return 'My list: {}'.format(', '.join(self.my_list))

    def in_my_list(self,item):
        if item in self.my_list:
            return 'Got it!'
        else:
            return 'Nope!'
    def snack_check(self):
        return self.snack in self.my_list
```

```
#My favorite snack is chocolate... But in january I'm going to pretend it's oranges
jan = My_list('apples')
jan.snack_check()
```

```
#But in February, I'm back to the default
feb = My_list()
feb.snack_check()
```

About that object...

```
dir(object)
```

These are all the things you inherit by subclassing object.

```
class caps_list(My_list):
    def in_my_list(self,item):
        response = super(caps_list,self).in_my_list(item)
        return response.upper()
```

```
shouty = caps_list()
```

```
shouty.in_my_list('chocolate')
```

```
dir(caps_list)
```

You can also call the super class directly, like so:

```
class caps_list(My_list):
    def in_my_list(self,item):
        # But you still have to pass self
        response = My_list.in_my_list(self,item)
        return response.upper()
```

```
shouty = caps_list()
shouty.in_my_list('chocolate')
```

Super actually assumes the correct things... Most of the time.

```
class caps_list(My_list):
    def in_my_list(self,item):
        response = super().in_my_list(item)
        return response.upper()
```

```
shouty = caps_list()
shouty.in_my_list('chocolate')
```

```
help(super)
```

Lesson 08: Modules, Namespaces, and Packages

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

0

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

3 2 464 207

python

(U//~~FOUO~~) A lesson on Python modules, namespaces, and packages for COMP3321.

Recommendations

~~UNCLASSIFIED//FOR OFFICIAL USE ONLY~~

(U) Modules, Namespaces, and Packages

(U) We have already been using modules quite a bit -- every time we've run `import`, in fact. But what is a module, exactly?

(U) Motivation

(U) When working in Jupyter, you don't have to worry about your code disappearing when you exit. You can save the notebook and share it with others. A Jupyter notebook kind of behaves like a python **script**: a text file containing Python source code. You can give that file to the python interpreter on the command line and execute all the code in the file (kind of like "Run All" in a Jupyter notebook):

```
$ python awesome.py
```

(U) There are a few significant limitations to sharing code in Jupyter notebooks, though:

1. what if you want to share with somebody who has python installed but not Jupyter?
2. what if you want to share *part* of the code with others (or reuse part of it yourself)?

3. what if you're writing a large, complex program?

(U) All of these *do* have native solutions in Jupyter:

1. convert the notebook to a script (File > Download as > Python)
2. copy-paste...?
3. make a big, messy notebook...?

(U) ...but they get unwieldy fast. This is where modules come in.

(U) Modules

(U) At its most basic, a **module** in Python is really just another name for a script. It's just a file containing Python definitions and statements. The filename is the module's name followed by a `.py` extension. Typically, though, we don't run modules directly -- we *import* their definitions into our own code and use them there. Modules enable us to write *modular* code by organizing our program into logical units and putting those units in separate files. We can then share and reuse those files individually as parts of other programs.

(U) Standard Modules

(U) Python ships with a library of standard modules, so you can get pretty far without writing your own. We've seen some of these modules already, and much of next week will be devoted to learning more about useful ones. They are documented in full detail in the Python Standard Library reference.

(U) An awesome example

(U) To understand modules better, let's make our own. This will put some Python code in a file called `awesome.py` in the current directory.

```
contents = '''
class Awesome(object):
    def __init__(self, awesome_thing):
        self.thing = awesome_thing
    def __str__(self):
        return "{0.thing} is awesome!!!".format(self)

a = Awesome("Everything")
print(a)
'''

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

(U) Now you can run `python awesome.py` on the command line as a Python script.

(U) Using modules: `import`

(U) You can also import `awesome.py` here as a module:

```
import awesome
```

(U) Note that you leave out the file extension when you import it. Python knows to look for a file in your path called `awesome.py` .

(U) The first time you import the module, Python executes the code inside it. Any defined functions, classes, etc. will be available for use. But notice what happens when you try to import it again:

```
import awesome
```

(U) It's assumed that the other statements (e.g. variable assignments, print) are there to help *initialize* the module. That's why the module is only run once. If you try to import the same module twice, Python will not re-run the code -- it will refer back to the already-imported version. This is helpful when you import multiple modules that in turn import the same module.

(U) However, what if the module changed since you last imported it and you really want to do want to re-import it?

```
contents = '''
class Awesome(object):
    def __init__(self, awesome_thing):
        self.thing = awesome_thing
    def __str__(self):
        return "{0.thing} is awesome!!!".format(self)

def cool(group):
    return "Everything is cool when you're part of {0}".format(group)

a = Awesome("Everything")
print(a)
'''

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

(U) You can bring in the new version with the help of the `importlib` module:

```
import importlib
importlib.reload(awesome)
```

(U) Calling the module's code

(U) The main point of importing a module is so you can use its defined functions, classes, constants, etc. By default, we access things defined in the `awesome` module by prefixing them with the module's name.

```
print(awesome.Awesome("A Nobel prize"))
```

```
awesome.cool("a team")
```

```
print(awesome.a)
```

(U) What if we get tired of writing `awesome` all the time? We have a few options.

(U) Using modules: `import ___ as ___`

(U) First, we can pick a nickname for the module:

```
import awesome as awe
```

```
print(awe.Awesome("A book of Greek antiquities"))
```

```
awe.cool("the Python developer community")
```

```
print(awe.a)
```

(U) Using modules: `from ___ import ___`

(U) Second, we can import specific things from the `awesome` module into the current *namespace*:

```
from awesome import cool
```

```
cool("this class")
```

```
print(Awesome("A piece of string"))  # will this work?
```

```
print(a)  # will this work?
```

(U) Get everything: `from ___ import *`

(U) Finally, if you really want to import *everything* from the module into the current namespace, you can do this:

```
from awesome import *  # BE CAREFUL
```

(U) Now you can re-run the cells above and get them to work.

(U) Why might you need to be careful with this method?

```
# what if you had defined this prior to import?
def cool():
    return "Something important is pretty cool"
```

```
cool()
```

(U) Get one thing and rename: `from ___ import ___ as ___`

(U) You can use both `from` and `as` if you need to:

```
from awesome import cool as coolgroup
```

```
cool()
```

```
coolgroup("the A team")
```

(U) Tidying up with `__main__`

(U) Remember how it printed something back when we ran `import awesome`? We don't need that to print out every time we import the module. (And really aren't initializing anything important.) Fortunately, Python provides a way to distinguish between running a file as a script and importing it as a module by checking the special variable `__name__`. Let's change our module code again:

```
contents = '''
class Awesome(object):
    def __init__(self, awesome_thing):
        self.thing = awesome_thing
    def __str__(self):
        return "{0.thing} is awesome!!!".format(self)

def cool(group):
    return "Everything is cool when you're part of {0}".format(group)

if __name__ == '__main__':
    a = Awesome("Everything")
    print(a)
'''

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

(U) Now if you run the module as a script from the command line, it will make and print an example of the `Awesome` class. But if you import it as a module, it won't -- you will just get the class and function definition.

```
importlib.reload(awesome)
```

(U) The magic here is that `__name__` is the name of the current module. When you import a module, its `__name__` is the module name (e.g. `awesome`), like you would expect. But a running script (or notebook) also uses a special module at the top level called `__main__`:

```
__name__
```

(U) So when you run a module directly as a script (e.g. `python awesome.py`), its `__name__` is actually `__main__`, not the module name any longer.

(U) This is a common convention for writing a Python script: organize it so that its functions and classes can be imported cleanly, and put the "glue" code or default behavior you want when the script is run directly under the `__name__` check. Sometimes developers will also put the code in a function called `main()` and call that instead, like so:

```
def main():
    a = Awesome("Everything")
    print(a)

if __name__ == '__main__':
    main()
```

(U) Namespaces

(U) In Python, *namespaces* are what store the names of all variables, functions, classes, modules, etc. used in the program. A namespaces kind of behaves like a big dictionary that maps the name to the thing named.

(U) The two major namespaces are the *global* namespace and the *local* namespace. The global namespace is accessible from everywhere in the program. The local namespace will change depending on the current *scope* -- whether you are in a function, loop, class, module, etc. Besides local and global namespaces, each module has its own namespace.

(U) Global namespace

(U) `dir()` with no arguments actually shows you the names in the global namespace.

```
dir()
```

(U) Another way to see this is with the `globals()` function, which returns a dictionary of not only the names but also their values.

```
sorted(globals().keys())
```

```
dir() == sorted(globals().keys())
```

```
globals()['awesome']
```

```
globals()['cool']
```

```
globals()['coolgroup']
```

(U) Local namespace

(U) The local namespace can be accessed using `locals()`, which behaves just like `globals()`.

(U) Right now, the local namespace and the global namespace are the same. We're at the top level of our code, not inside a function or anything else.

```
globals() == locals()
```

(U) Let's take a look at it in a different scope.

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

(U) You can access submodules by chaining them together with dot notation:

```
import sound.effects.reverse
```

(U) The other methods of importing work as well:

```
from sound.filters import karaoke
```

(U) `__init__.py`

(U) What is this special `__init__.py` file?

- (U) Its presence is required to tell Python that the directory is a package
- (U) It can be empty, as long as it's there
- (U) It's typically used to initialize the package (as the name implies)