(U)

Instructor Notes

0

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

3 144 6

python fcs6

(U//FOUO) Instructor notes for COMP 3321.

Recommendations

UNCLASSIFIED//~~FOR OFFICIAL USE ONLY~~

(U) So, you're teaching the Python class. What have you gotten yourself into? You should probably take a few moments (or possibly a few days) to reconsider the life choices that have put you in this position.

(U) Course Structure

(U) As mentioned in the introduction, this course is designed for flexibility. When taught in a classroom setting, a single lesson or module can be covered in a session that lasts between 45 and 90 minutes, depending on the topics to be covered. The standard way to structure the course is as a full-time, two week block. During the first week, the ten lessons are covered with morning and afternoon lectures. During the second week, up to ten modules are covered in a similar manner, as needed or requested by the students in the class. (If the class needs are not known, take a vote). During the first few days of class, students should choose a project to work on. On the last day, students should report back on their progress and, if possible, demonstrate their work. Instructors should be available outside of lectures to assist students with exercises and projects.

(U) The two week block is not the only way of teaching the course. The material could be presented at a more leisurely pace, for instance during a weekly brown bag lunch that continues for several months. Alternatively, if students are already prepared (or willing to do some of the initial lessons in a self-study manner), a great deal can be accomplished in a two or three day workshop. For instance, if all students already have a basic knowledge of Python, they might well start with the lessons on tooling and writing modules and packages, then move on to cover various modules of interest.

(U) Instructional Style

(U) When teaching mathematics, the common practice of the instructor writing solutions on the chalkboard is a moderating method that helps students keep up. Writing on a chalkboard is not usually helpful when teaching programming, but the same principle applies; as the instructor, you should adopt practices that help you **slow down**. We recommend that you have a live, interactive session displayed at the front of the room, large enough for all the students to see. This session can either be a terminal session or a Jupyter notebook. The important detail is that in most cases the commands should not be pre-populated; e.g. you should not just execute cells from an existing Jupyter notebook. The materials are present to help you prepare, and as a reference for the students as they work on exercises; they are not an acceptable substitute for the shared experience of teaching and learning. You will make unexpected mistakes as you write code live in front of the class. Don't worry, relax, and let the students help you--it will help them learn the principles and figure out how to solve their own problems. **This is not a substitute for proper preparation**; too many mistakes and fumbles will cause your students to lose interest in the course and trust in you.

(U) Ongoing Development

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

(U//~~FOUO~~) The developers of this course believe that the current materials are sufficiently well developed to be an effective aid for the course. However, improvements, extensions, and refinements are always welcome. To that end, we have attempted to make it easy to contribute to the project. The documentation is based on the NSAG fork of [redacted] original COMP 3321 materials. If you want to make changes for your own purposes, feel free to clone that repository or fork any of these notebooks on the Juptyer Gallery. Please submit a change request on the Gallery if you'd like to make a one-off contribution, including new or improved exercises, additions to lessons or modules, or entirely new lessons or modules. If you would like to be a collaborator on all of the COMP 3321 notebooks, contact the COMP3321 GlobalMe group and ask to be added.

(U) A possible icebreaker

(U) To get the students interacting with each other as well as thinking about code at an abstract level, consider the following icebreaker. Instructor becomes human compiler to interpret written instructions to get out of the room.

(U) **Phase I** -- discussion with whole class

(U) Invent a programming language together, one sufficient for this task. Clearly explain the task, showing where in the room the instructor will begin and how big a step is.

(U) Take suggestions from students about what instructions they will want to use. Write each instruction on the board. Be clear that everything written on the paper must come from syntax on the board. No other syntax allowed.

(U) Make sure the instructor and students agree about precisely what each instruction means. As the instructor, be certain the list on the board is enough for you to solve the problem. Give hints until it's complete.

(U) Let the students come up with the syntax. But here are some syntax examples they may come up with.

- step(n) -- takes in an integer and causes instructor to take n steps.

- turn(d) -- takes in a number in degrees and caused instructor to turn clockwise that many degrees. Students may abuse this and put in numbers that lead to dizzyness, e.g. turn(1440)
- obstacle -- returns boolean indicating rather an obstacle is directly in front of instructor. Variations for checking to the left or right may be desirable as well.
- if < > then < > else < > -- first blank takes boolean (make sure boolean functions exist!). Second blanks take instructions.
- while < >: < > -- takes boolean function and any expression
- not -- expression to reverse a boolean
- any integer

(U) **Phase II** -- break into teams

(U) Teams of 3-5 students tend to be appropriate.

(U) Each team produces a piece of paper with computer instructions for the human compiler to get out of the room. Only allowable syntax is what is written on the board. Should take no more than 15 minutes.

(U) **Phase III** -- demonstrations

(U) One at a time, the instructor takes a team's solution and follows the instructions, literally and fairly. Does the instructor get out of the room?

(U) Introductory e-mail

(U) There are a couple things it would be nice if the students could have done in advance, in particular having GITLAB accounts and INHERE agreements. I have the COMP3321 learning facilitators send the following e-mail to enrolled students:

(U) Aloha--

(U) You are receiving this e-mail because you are registered for COMP3321 beginning <..>.

(U) If at all possible, we could use you do a few setup things in advance to make class go smoothly on the first day. Really, two simple things that will take a minute of your time and save us hours on the first day of class.

(U) 1) GO iagree and find, read and agree to the INHERE user agreement.

(U) 2) GO gitlab.

(U) In more detail:

(U) 1) The course will be run via LABBENCH and NBGALLERY. Before using this, you will need to agree to the terms of service and [redacted] acknowledge this. Instructions can be found here: https://nbgallery.nsa.ic.gov/ (GO NBGALLERY) by clicking GET A BENCH. We only need you to follow the first step, outlined below. Be careful going further because LABBENCH machines self-destruct two weeks after creation, so you'll want a fresh one the first day of class.

(U) GO iagree. Search "inhere". Read the INHERE user agreement and agree to it. [redacted] will eventually acknowledge this agreement. That's what we need.

P.L. 86-36

P.L. 86-36

(U) 2) GO gitlab. By going there once, an account will be created. That's all we need before we start.

(U) If you want to go further and use git from the command line [redacted] follow instructions here: https://wiki.nsa.ic.gov/wiki/Git/Windows

(U) There are several options presented. Whatever you can get to work is fine.

(U) 3) Optional. The class will be run off LABBENCH. But some people prefer to use [redacted] machine. This will take some setup work. Install Anaconda [redacted] by following these instructions: https://production.tradecraft.proj.nsa.ic.gov/entry/29475

(U) If you have any issues with the instructions, please contact the instructor: <..>

(U) Submitting projects

(U) A possible avenue for submitting of projects is GITLAB. [redacted] The instructor adds all students in the class to the Gitlab group comp3321[redacted] A new project is created within that group entitled class-projects-Mmm-YYYY. Students will submit their code to Gitlab. This can be easily accomplished through the web application by finding the '+' sign and copying and pasting code. The web application does not allow for the creation of folders. Students who need folders can make them from the command line or get help doing so.

Python Programming

0

Updated 3 months ago by [redacted] in COMP 3321

Python3 thumbnail 52 3185 309

python comp3321

(U//~~FOUO~~) Course introduction and syllabus for COMP 3321, Python Programming.

Recommendations

UNCLASSIFIED//~~FOR OFFICIAL USE ONLY~~

(U) History

> ...in December, 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office ... would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendent of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of *Monty Python's Flying Circus*)
>
> Guido van Rossum, *Foreword for Programming Python, 1st Edition.*

(U) Motivation

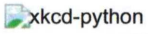
(U) Python was designed to be easy and intuitive without sacrificing power, open source, and suitable for everyday tasks, with quick development times. It makes the layers between *programming* and *problem solving*seem as thin as possible. It's suitable for:

- Opening an interactive session to solve the Daily Puzz,
- Writing a script that automates a tedious and time-consuming task,

- Creating a quick web service or an extensive web application, and
- Doing advanced mathematical research.

(U) If you don't know any programming languages yet, Python is a good place to start. If you already know a different language, it's easy to pick Python up on the side. Python isn't entirely free of frustration and confusion, but hopefully you can avoid those parts until long after you get some good use out of Python.

(U) Programming is not a spectator sport! The more you practice programming, the more you will learn in this class, both in breadth and depth. Python practically teaches itself--the goal of your instructors is to guide you to the good parts and help you move just a little bit more quickly than you would otherwise. *Happy Programming!*



(U) Objective

(U) The goal of this class is to help students accomplish work tasks more easily and robustly by programming in Python. To pass the course, each student must write at least one Python program that has substantial personal utility or is of significant personal interest. When choosing a project, students are encouraged to first think of work-related tasks. For students who need help getting started, several suggestions for possible projects are found at the bottom of this page. On the first day, instructors will lead a discussion where project ideas are discussed.

(U) This class is designed for students of varying backgrounds and levels of experience, from complete novice to competent programmer. Asking each student to design and implement their own project allows everyone to learn and progress at an individual pace.

(U) Logistics

(U//~~FOUO~~) This course is designed to be suitable for self-learning. Even if no formal offerings are available for your schedule, you may access and work on the modules of this course at any time. Even if you don't have access to a recent version of Python on a workstation or virtual machine, you can access a personalized Jupyter notebook available on LABBENCH. For an individual pursuing this self-study option, it is recommended to first cover the **Python Basics** roughly in order, then select as many of the **Useful Modules** as seem appropriate. You can also use this Jupyter notebook to experiment and write solutions to exercises.

(U//~~FOUO~~) One possibility for a group of potential students who start out with a different amounts of programming experience is to use Jupyter as a self-study tool until everyone has a basic understanding of programming, then follow up with an abbreviated instructor-led course (anywhere from two days to a week or more, depending on needs).

(U) In the Classroom

(U) For a two week course: there will be a morning lecture and an afternoon lecture every day. The morning lecture will last between an hour and ninety minutes; the afternoon lecture will be somewhat shorter. **If a lecture is going too fast, please ask questions to slow us down!** If it's going too slow, feel free to work ahead on your own.

Doc ID: 6689692

(U//~~FOUO~~) You will either use Python within [redacted] environment or within LABBENCH. [redacted] While we will point out some differences between the Python 3.x and 2.x lines, this course will focus on Python 3. If you need or want to run Python on Linux, probably within a MachineShop VM, we'll work with you. To the extent possible, we will write code in platform-agnostic manner and point out features that are unique to specific versions of Python.

(U) Although lectures will only take up two or three hours each day, we encourage you to spend the remainder of your day programming in Python, either on your own or in groups, but in the classroom if possible. At least one instructor will be available in the classroom during normal business hours.

(U) This course is a work in progress; we welcome all suggestions. There are more **Useful Modules** than we can hope to cover in a two-week class; instructors will take a vote to determine which modules to cover. If there is another topic that you would like to have covered, especially along the lines of "How would I do `X`, `Y`, or `Z` in Python?", please ask--if there's enough interest, we'll cover it in a lecture. We'd even be happy to have you contribute to the course documentation! Talk to an instructor to find out how.

P.L. 86-36

(U) Table of Contents

(U) Part I: Python Basics (Week 1)

Doc ID: 6689692

(U) Part II: Useful Modules (Week 2)

- (U) Under Construction Module: Collections and Itertools
  - (U) Supplement: Functional Programming
  - (U) Supplement: Recursion Examples
- (U) Under Construction Module: Command Line Arguments
- (U) Module: Dates and Times
  - (U) Datetime Exercises
- (U) Module: Interactive User Input with ipywidgets
- (U) Module: GUI Basics with Tkinter
  - (U) Supplement: Python GUI Programming Cookbook
- (U) Under Construction Module: Logging
- (U) Under Construction Module: Math and More
  - (U) Supplement: COMP3321: Math, Visualization, and More!
- (U) Module: Visualization
- (U) Module: Pandas
- (U) Under Construction Module: A Bit About Geos
- (U) Under Construction Module: My First Web Application
- (U) Under Construction Module: Network Communication Over HTTPS and Sockets
  - (U) Supplement: HTTPS and PKI Concepts
  - (U//~~FOUO~~) Supplement: Python, HTTPS, and LABBENCH
- (U) Module: HTML Processing with BeautifulSoup
- (U) Under Construction Module: Operations with Compression and Archives
- (U) Under Construction Module: Regular Expressions
- (U) Under Construction Module: Hashes
- (U) Under Construction Module: SQL and Python
  - (U) Supplement: Easy Databases with sqlite3
- (U) Module: Structured Data: CSV, XML, and JSON
- (U) Module: System Interaction
  - (U) Supplement: Manipulating Microsoft Office Documents with win32com
- (U) Module: Threading and Subprocesses
- (U//~~FOUO~~) Distributing a Python Package at NSA
- (U) Module: Machine Learning Introduction

(U) Homework

- (U) Day 1 Homework

- (U) Day 2 Homework

(U) Exercises (with Solutions)

- (U) Dictionary and File Exercises
- (U) Structured Data and Dates
- (U) Datetime Excercises
- (U) Object-Oriented Programming and Exceptions

(U) Class Projects

(U) Click here to get to a notebook containing instructions for password checker and password generator projects.

(U) Project Ideas

- (U) Write a currency conversion script
- (U) Write a web application
- (U) Do Project Euler problems
- (U) Find Anomalous Activity on the BigCorp Network
- (U) RSA Encryption Module, Part 2
- (U) Pick a project from one of the Safari books below

(U) General Resources

(U) Python Language Documentation

- (U) Python 2.7.10
- (U) Python 2.7 on DevDocs
- (U) Python 3.4.3
- (U) Python 3.5 on DevDocs

(U//~~FOUO~~) NSA Course Materials

- (U) Instructor Notes
- (U//~~FOUO~~) COMP 3321 Learn Python server
- (U//~~FOUO~~) CRYP 3320 (CES version of the course)
- (U//~~FOUO~~) [redacted] COMP 3320 materials
- (U//~~FOUO~~) CADP Python Class

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

(U) Books

- (U) Automate the Boring Stuff with Python
- (U) Black Hat Python: Python Programming for Hackers and Pentesters
- (U) Dive into Python
- (U) Expert Python Programming
- (U) Head First Python
- (U) High Performance Python (advanced)
- (U) Learning Python
- (U) Learning Python Programming (videos)
- (U) Programming Python
- (U) Python Crash Course (includes 3 sample projects)
- (U) Python Playground (more sample projects)
- (U) Python Pocket Reference (consider getting a print copy)
- (U) Python Programming for the Absolute Beginner (even more sample projects, games & quizzes)
  - (U) and More Python Programming for the Absolute Beginner
- (U) Think Python
- (U) Safari Books (General Query)

(U) Other

- (U) Final Project Schedule Generator
  - (U) Just a little notebook for randomly generating a schedule for students to present their final projects for COMP3321.
- (U//~~FOUO~~) The Python group on NSA GitLab
- (U) The Hitchhiker's Guide to Python!
- (U//~~FOUO~~) Python on WikiInfo
- (U) Python on StackOverflow

(U) Additional targeted resources (often excerpts from the above) are linked in each lesson and module.

Lesson 01: Introduction: Your First Python Program

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

0

Updated 8 months ago by [redacted] in COMP 3321

3 24 2693 747

fcs6 python

(U) Covers Anaconda installation, the python interpreter, basic data types, running code, and some built-ins.

Recommendations

UNCLASSIFIED//~~FOR OFFICIAL USE ONLY~~

(U) Welcome To Class!

(U) Let's get to know each other. Stand up and wait for instruction.

(U) Who has a specific project in mind?

(U) Method 1: Anaconda Setup

(U) Alternately, follow this tradecraft hub article. https://production.tradecraft.proj.nsa.ic.gov/entry/29475

(U) We will be using version 4.4.0 of the Anaconda3 Python distribution, available from Get Software. Anaconda includes many packages for large-scale data processing, predictive analytics, and scientific computing.

(U) Installation Instructions

Doc ID: 6689693

1. (U) Click on the link above
2. (U) Click on the "Download Now" button
3. (U) Accept the agreement and click "Next"
4. (U) Click "Next"
5. (U) Download "Anaconda3-4.4.0-Windows-x86_64.exe
6. (U) Open the folder containing the download (typically this is your "Downloads" folder)
7. (U) Doubleclick the Anaconda3-4.4.0-Windows-x86_64.exe file
8. (U) Click "Next" to Start the installer
9. (U) Click the "I Agree" button to accept the license agreement
10. (U) Choose to install for "Just Me" and click "Next"
11. (U//~~FOUO~~) Select a destination folder on your U: drive (such as U:\private\anaconda3)
    1. (U//~~FOUO~~) Click the "Browse..." button, Click on "Computer" and select the U: drive
    2. (U//~~FOUO~~) Select "My Documents" and press "OK" [Note: DO NOT Make a folder named anaconda3]
    3. (U//~~FOUO~~) In the Destination Folder input area add "anaconda3" as the folder name
    4. (U//~~FOUO~~) Click "Next"
12. (U) Click "Install" to begin the install (leave checkboxes as is)
13. (U) Wait about 30 minutes for the install to complete

P.L. 86-36

(U) Running python

(U) You can run python directly on your [redacted] desktop and immediately interact with it:

1. (U) Open a Windows command window (Type "cmd" in the Windows Programs search bar)
2. (U) Type "python" in the command window

(U) Running Jupyter

(U) Alternately, you can run python in a browser from a web-enabled python, called a jupyter notebook. [redacted] The Notebook Gallery is at go nbgallery. For this class, however, we'll each start up our own individual Jupyter web-portal to run our class notebooks.

1. (U) From the Windows Start menu, search for "jupyter"
2. (U) Right-click on Jupyter Notebook in the results and select Properties
3. (U//~~FOUO~~) In the "Target" field, add " u:\private" at the end (after "notebook") [Note: don't forget the space before u:\private]
4. (U) Click Apply and then OK
5. (U) Search for jupyter again in the start menu and click on Jupyter Notebook to run it
6. (U) Wait a few moments...This should launch Jupyter in your browser at http://localhost:8888/tree

(U) Method 2: LABBENCH Setup

Doc ID: 6689693

(U//~~FOUO~~) Step 1: Access to LABBENCH

- (U//~~FOUO~~) go iagree and read and accept the INHERE User Agreement.

(U//~~FOUO~~) This is a prerequisite for access to LABBENCH, a VM system where we will be working. It may take a few hours for the approval to propagate through the system.

(U) Step 2: Visit Jupyter Gallery

1. (U//~~FOUO~~) go jupyter
2. (U) Click on the Jupyter Gallery logo to get to the Gallery.
3. (U) Click on Tour the Gallery for quick demo.
4. (U//~~FOUO~~) To find the course notebooks, either search for "Syllabus" or choose Notebooks > Learning > COMP 3321 and sort by title to find the Syllabus.

(U//~~FOUO~~) Step 3: Set up Jupyter on LABBENCH

(U//~~FOUO~~) At the Jupyter Gallery, click "Jupyter on LABBENCH" for a tutorial on how to get set up.

(U) Basic Basics: Data and Operations

(U) The most basic data types in Python are:

- Numbers
  - Integer `<type 'int'>` (these are "arbitrary precision"; no need to worry whether it's 32 bits or 64 bits, etc.)
  - Float `<type 'float'>`
  - Complex `<type 'complex'>` (using `1j` for the imaginary number)
- Strings `<type 'str'>`
  - No difference between single and double quotes
  - Escape special characters (e.g. quotation marks)
  - Raw string `r'raw string'` prevents need for some escapes
  - Triple-quotes allow multiple line strings
  - Unicode `u'Bert \x26 Ernie'` `<type 'unicode'>`
- Booleans: `True` and `False`

(U) We operate on data using

- **operators**, e.g. mathematical operators `+`, `-`; the keyword `in`, and others

- **functions**, which are operations that take one or more pieces of data as arguments, e.g. `type('hello')`, `len('world')`, and
- **methods**, which are attached to a piece of data and called from it using a `.` to separate the data from the method, e.g. `'Hello World'.split()`, or `'abc'.upper()`

(U) Deep in the guts of Python, these are all essentially the same thing, but syntactically and pedagogically it makes sense to separate them.

(U) Pieces of basic data can be stored inside containers, including

- Lists
- Dictionaries
- Sets

but we'll introduce those later.

(U) The Interactive Interpreter

With that basic background, let's try some things in your Windows command window...

```
U:\private>python
Python 3.5.1 |Anaconda 2.5.0 (64-bit)| (default, Jan 29 2016, 15:01:46) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
```

```
5 + 7
```

```
type(5+7)
```

```
4.5 - 5.5
```

```
type(7.1 - 2.1)
```

```
13/5            # this changed in python3
```

```
13//5
```

```
1j * 4j
```

```
"hello" + " world"
```

```
"hello " * 10
```

(U) Executing code in a file

Open the file `first-program.py` (or anything ending with `.py`) in your favorite editor (I use `emacs`, but you can use whatever you want).

(U) If you don't have a favorite editor do this:

1. (U) Go to your Jupyter portal at http://localhost:8888/tree
2. (U) Pull down the "New" button menu and choose "Text File"
3. (U) Click on the "Untitled1.txt" name and enter the new file name as "first-program.py"

(U) Type some Python statements in it:

```
5+7
9*43
8**12
```

(U) Don't forget to save it (File->Save from Jupyter).

(U) To run it, give the file name as an argument to Python:

Make sure the command window is referencing the same folder as the file. That is U:\private for most.
If your command window is not referencing U:\private, do this:
1. Enter "U:"
2. Enter "cd private"

```
U:\private>python first-program.py
```

(U) Nothing appears to happen, because auto-printing of the output of a function *only happens in the interpreter*. Fix it up:

```
print(5+7)
print(9*43)
print(8**12)
```

(U) Built-in functions and methods

(U) Some functions work on almost any arguments that you supply:

- `help(x)` : shows interactive help
- `dir(x)` : gives the **directory** of the object, i.e. all the methods available
- `type(x)` : tells you the type of `x` —a type is almost the same as any other object
- `isinstance(a,b)` : tells if object `a` is an instance of `b`, which must be a `type`; something like `type(a) == b`
- `print`
- `hasattr(a,b)` : tells whether `a` has something by the name `b`; something like `b in dir(a)`
- `getattr`

- `id`
- `input`

(U) Constructor functions usually try to do their best with the argments you give, and return the appropriate data of the requested type:

- `str` : turns numbers (and other things) into their string representations
- `int` : truncates `float` , parses `str` ings containing a single integer, with optional **radix (i.e. base)**, error on `complex`
- `float` : parses `str` ings, gives `float` representation of `int` , error on `complex`
- `complex` : takes `(real,imag)` numeric arguments, or parses a `str` for a single number

(U) Other functions only work with one or two types of data:

- Numbers:
  - Functions: `abs` , `round` , `float` , `max` , `min` , `pow` (modular), `chr` , `divmod` , etc.
  - Operators: Standard math, bitwise: `<<` , `>>` , `&` , `|` , `^` , `~`
  - Methods: Numeric classes don't have methods
- Strings:
  - Functions: `len` , `min` , `max` , `ord`
  - Operators: `+` , `*` (with a number), `in`
  - Methods: `strip` , `split` , `startswith` , `upper` , `find` , `index` , many more; use `dir('any string')` to find more

(U) Exercises:

1. Make a shopping list of five things you need at the grocery store. Put each item on it's own line in a cell. Remember to use quotes! Use print() so that each of your items displays (try it first without).
2. Your groceries ring up as 9.42, 5.67, 3.25, 13.40, and 7.50 respectively. Use python as a handy calculator to add up these amounts.
3. But wait! You decide you to buy five of the last item. Re-calculate your total.
4. Using the `len()` function, determine the number of characters in the string "blood-oxygenation level dependent functional magnetic resonance imaging" (Fun fact: this string is the longest entry in WordNet3.1 Index).
5. Pick your favorite snack. Use the `*` operator to print 100 copies of it. Modify your code to have them print with spaces between them.
6. Challenge: Run `dir('any string')` . Pick two methods that sound interesting and run `help('any string'.interesting_method)` for both of them. Can you figure out how to use these methods?
7. Bonus challenge: Can you figure out how to get the same output as Exercise 1 using only one print statement? If so, can you also do it in one line of code?

Lesson 02: Variables and Functions

0

Updated 5 months ago by [redacted] in COMP 3321

3 1196 473

python fcs6

(U) Introduction to variables and functions in Python.

Recommendations

UNCLASSIFIED

(U) My Kingdom for a Variable

(U) All the numbers and strings in the world won't do you any good if you can't keep track of them. A **variable** in Python is a name that's attached to something--a number, a string, or something more complicated, like a list, an object, or even a type.

(U) Python is dynamically typed, which means that a variable, once declared, can hold different types of data over its lifetime. A variable is declared with the `=` operator. Go ahead, give that value a name!

```
x = 2
```

```
x
```

```
x + 5
```

(U) In interactive mode, a `_` is a reference to the last cell output.

```
9*8
```

```
_
```

(U) This can be especialy helpful when you forget to save the output of a long computation by giving it a name.

```
y = x + 5
```

```
_
```

(U) Notice the line `y = x + 5` produced no output, so was ignored when `_` was called.

```
isinstance(x, int)
```

```
isinstance(x, str)
```

(U) So, let's change what `x` is equal to (and even change its type!) by just reassigning the variable name.

```
x = "Hello"
```

```
isinstance(x, int)
```

```
isinstance(x, type(8))
```

```
isinstance(x, type('a'))
```

```
y = x + 5
```

(U) So what about converting from one type to another?

```
a = "3.1234"
```

```
type(a)
```

```
b = float(a)
```

```
type(b)
```

```
b
```

```
float(x) # This should fail. Why?
```

```
c = str(b)
```

```
c
```

```
i = int(b)
```

```
i
```

(U) Go ahead and use the `dir()` function to see what variables you have defined. This commands shows all the objects that are defined in your current scope (we will talk about scope later).

```
dir()
```

```
del x
```

```
dir()
```

```
x + 5 # Why does this crash?
```

(U) We can also assign variables with some fancy shortcuts:

```
a = b = c = 0
```

```
print(a)
print(b)
print(c)
```

```
x, y = 1, 2
```

```
print(x)
print(y)
```

```
z = 1, 2        # What does this do??
z
```

```
x, y, z = 1, 2  # How about this?
```

Note that since the last command failed, the values of x, y, and z were unchanged.

```
print(x)
print(y)
```

```
x, y = y, x     # Fast Swapping!
```

```
print(x)
print(y)
```

(U) Variable names can be assigned to all the different object types. Keep these tricks in mind as you learn about more complex types.

(U) Let's talk lists for a minute. We'll go into details about containers later in the course, but you'll need to know the basics for one of the exercises.

```
l = [1, 2, 3, 4]
```

```
1 in l
```

```
5 in l
```

```
l = ["one", "two", "three", "four"]
```

```
"one" in l
```

(U) Exercises

1. Save a copy of your favorite snack in a variable. Using that variable, print your snack a 100 times.
2. Ask your neighbor what their favorite snack is. Save it in a variable. You should now have two variables containing snacks. Add (concatenate) them together and print the result 100 times.
3. Using the `[]` notation above, make a list of five groceries and save in a variable. (If you did the earlier grocery list exercise, use those items). Using the variable from Exercise 1, test to see if your favorite snack is " `in` " the list.
4. Using your grocery list from Exercise 3, and the variable from Exercise 2, test to see if your neighbor's favorite snack is on your list just as you did for your snack.
5. Use the "fast swapping" to swap your favorite snack with your neighbor's. Print both variables to see the result. Are you happy or sad with your new favorite snack?

(U) Functions

(U) So what else can we do with variables? Lots!

```
7 % 2     # Modulo operator
```

```
7 ** 2
```

```
min(2, 7) # built-in function
```

```
max(2, 7)
```

```
dir("a")
```

(U) Python comes with a bunch of built-in functions. We've used a few of these already: `dir()`, `min()`, `max()`, `isinstance()`, and `type()`. Python includes many more, such as:

```
abs(-1)
```

```
round(1.2)
```

```
len("12345")
```

(U) But functions take memory, and there are hundreds of modules included with Python, so we can't have access to everything that Python can do all at once. In order to use functions that aren't built in, we must tell Python to load them. We do this with the `import` statement:

```
import os
```

(U) This loads the **os** module. A **module** is a file containing definitions and statements, and are generally used to hold a collection of relatated functions. The **os module** contains functions relating to the Operating System of the computer where Python is running.

(U) So what's contained in `os` ? Let's look:

```
dir(os)
```

(U) That gives you a list of everything defined in the os module.

```
os.name      # why doesn't name require parentheses?
```

```
os.listdir()
```

(U) Python has robust documentation on the standard modules. Always consult the documentation if you are unsure how to use a function.

(U) What if I don't need everything in a module?

```
from os import listdir
```

```
listdir()
```

(U) We'll get into more modules later in the class. For now we'll just touch on two others:
`sys` , which contains variables and functions relating to Python's interaction with the system; and `random` , which provides random number generation.

```
import sys
```

```
dir(sys)
```

```
sys.argv    # holds command line arguments
```

```
sys.exit()  # exits Python (you may not want to type this)
```

```
import random
```

```
random.randint(1, 5)
```

```
random.random()
```

(U) Exercises

1. Make a list of your grocery prices (9.42, 5.67, 3.25, 13.40, and 7.50 respectively) and store in a variable. Use built in functions to find the price of the cheapest and most expensive item on your grocery list.
2. `import random` and run `help(random.randint)` . Use `randint` to randomly print between 0 and 100 copies of your favorite snack.
3. Run `dir(random)` . Find a function in `random` that you can use to return a random item from your grocery list. Remember you can use `help()` to find out what different functions do!
4. Write code to randomly select a price from your list of grocery prices, round to the nearest integer, and print the result.
5. Challenge: Your grocery store is having a weird promotion called "win free change" ! A random item from your (price) list is chosen and you pay 10 dollars. If the item is less than 10 dollars you get your item and the change back as normal; however, if you get lucky and the price is more than 10 dollars you get the item and the difference in price back as change. Write code randomly pick a price from your price list and print out the amount of change the cashier has to pay you during this promotion. Hint: use the built in `abs` function.

(U) Making your own functions

(U) Functions (in Python) are really just special variables (or data types) that can have input and output. Once defined, you can treat them like any other variables.

(U) Functions are defined with a specific syntax:

- Start with the keyword `def` ,
- followed by the function name, and
- a list of arguments enclosed in `()` , then
- the line ends with a `:` , and
- the body of the function is indented on following lines.

(U) Python uses white space to determine blocks, unlike C, Java, and other languages that use `{}` for this purpose.

(U) To have output from the function, the `return` keyword is used, followed by the thing to be returned. For no output, use `return` by itself, or just leave it out.

```
def first_func(x):
    return x*2
```

```
first_func(10)
```

```
first_func('hello')
```

(U) Wow...Python REALLY does not care about types. Here is the simplest function that you can write in Python (no input, no output, and not much else!):

```
def simple():
    pass    # or return
```

```
simple()    # BORING!
```

(U) Let play around a bit with a new function...we shall call this powerful function `add`.

```
def add(a, b):
    return a+b
```

```
add(2, 3)
```

```
add(1)
```

```
add('a',3)
```

```
add('a','b')
```

```
add
```

```
def add2(a, b):
    print (a+b)
```

```
x = add(2, 3)  # What did this do?
x
```

```
x = add2(2, 3) # What did this do?
x
```

(U) **Don't forget**: function names are variables too.

```
x = add  # What did this do?
```

```
add = 7  # And this?
```

```
add(2,3) # We broke this function. A lesson here.
```

```
x(2,3)
```

(U) Exercises

1. Write an `all_the_snacks` function that takes a `snack` (string) and uses the `*` operator to print it out 100 times. Test your function using each of the items on your grocery list. What happens if you enter a number into your function? Is the result what you expected?
2. You may have noticed that your `all_the_snacks` function prints all your snacks squished together. Rewrite `all_the_snacks` so that it takes an additional argument `spacer`. Use `+` combine your snack and spacer before multiplying. Test your function with different inputs. What happens if you use strings for both `snack` and `spacer`? Both numbers? A string and an integer? Is this what you expected?
3. Rewrite `all_the_snacks` so that it also takes a variable `num` that lets you customize the number of times your snack gets printed out.
4. Write an `in_grocery_list` function that takes in a `grocery_item` returns `True` or `False` depending on whether the item is on your list.
5. Write a `price_matcher` function that takes no arguments, but prints a random grocery item and a random price from your price list every time it is run.
6. Challenge: modify your `price_matcher` to `return item, price` rather than print them. Write a `free_change` function that calls your new `price_matcher` and uses the result to print your item and the absolute value of the change for the item assuming you paid $10.

(U) Arguments, Keyword Arguments, and Defaults

```
def f(a, b, c):
    return (a + b) * c
```

(U) You can give arguments default values. This makes them optional.

```
def f(a, b, c=1):
    return (a + b) * c
```

```
f(2, 3)
```

```
f(2, 3, 2)
```

(U) You can call arguments by name also.

```
f(b=2, a=5)
```

```
f(b=2, c=5)
```

```
def g(a=1, b, c):
    return (a + b) * c  # What happens here?
```

(U) Exercises

1. Rewrite `all_the_snacks` so that `num` amd `spacer` have defaults of `100` and `', '` respectively. Using your favorite snack as input, try running your function with no additional input.
2. Try running `all_the_snacks` withyour favorite snack and the spacer `'! '` and no additional inputs. How would you run it while inputing your favorite snack and 42 for `num` while keeping the default for `spacer`? Can you use this method to enter `spacer` and `num` in reverse order?

Doc ID: 6689693

(U) Scope

(U) In programming, **scope** is an important concept. It is also very useful in allowing us to have flexibility in reusing variable names in function definitions.

```
x = 5
```

```
def f():
    x = 6
    print(x)
```

```
x
```

```
f()
```

```
x
```

(U) Lets talk about what happened here. Whenever we try to get or change the value of a variable, Python always looks for that variable in the most appropriate (closest) scope.

(U) So, in the function above, when we declared `x = 6`, we were declaring a local variable in the definition of `f`. This did not alter the global `x` outside of the function.
If that is what you want to happen, just use the `global` keyword.

```
x = 5
```

```
def f():
    global x
    x = 6
```

```
x
```

```
f()
```

```
x
```

(U) Be careful with scope, it can allow you to do some things you might not want to (or maybe you do!), like overriding built-in functions.

```
len('my string is longer than 3')
```

```
def len(x):
    return 3
```

```
len('my string is longer than 3')
```

(U) input

(U) The `input` function is a quick way to get data accessed from stdin (user input). It takes an optional string argument that is the prompt to be issued to the user, and *always* returns a string. Simple enough!

```
a = input('Please enter your name: ')
```

```
a
```

(U) Advanced Function Arguments

(U) Most of the time, you know what you want to pass into your function. Occasionally, it's useful to accept arbitrary arguments. Python lets you do this, but it takes a little bit of syntactic sugar that we haven't used before.

- List and dictionary unpacking
- List and dictionary packing in function arguments

(U) Exercises

1. Use `input` to ask for your favorite color and store it in the variable `my_color`. Use `input` to ask for your neighbor's favorite color and store it in the variable `neighbor_color`.
2. Use `input` to ask for your favorite number and store it in the variable `my_num`. Run `2 + my_num`. Why does this fail? How can you fix it?
3. Write a "April fool's" `color_swapper` function that takes `my_color` and and `neighbor_color` as inputs and prints a message declaring what your and your neighbor's favorite colors are respectively. Add a line before the print that swaps the contents of the variables so that now message is printed with your favorite colors swapped. Run your function and then print the contents of `my_color` and `neighbor_color`. How were you able to swap them in the function without swapping them in your notebook?
4. Challenge: Write a `global_color_swapper` that swaps your colors globally. Run your function and then print the contents of `my_color` and `neighbor_color`. Why might this be a bad idea, even for an April fool's joke?

(U) Review

1. Write a function called 'Volume' which computes and returns the volume of a box given the width, length, and height.
2. Write a function called 'Volume2' which calculates the box volume, assuming the height is 1, if not given.
3. Challange: Import the 'datetime' module. Experiment with the different methods. In particular, determine how to print the current time.

Lesson 2 - Function Exercises

0

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

3 355 91

python exercises

(U) Function Exercises for COMP3321 Lesson 2

Recommendations

(U) Lesson 2 - Functions Exercises

(U) Write a function `isDivisibleBy7(num)` to check if a number is evenly divisible by 7.

```
>>> isDivisibleBy7(21)
True
>>> isDivisibleBy7(25)
False
```

(U) Write a function `isDivisibleBy(num,divisor)` to check if num is evenly divisible by divisor.

```
>>> isDivisibleBy(35,7)
True
>>> isDivisibleBy(35,4)
False
```

(U) Make a function `shout(word)` that accepts a string and returns that string in capital letters with an exclamation mark.

```
>>> shout("bananas")
'BANANAS!'
```

(U) Make a function `introduce()` to ask the user for their name and shout it back to them. Call your function shout to make this happen.

```
>>> What's your name?
>>> Bob
HI BOB!
```

Lesson 2 - Variable Exercises

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

0

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

3 369 126

python exercises

(U) Variable Exercises for COMP3321 Lesson 2

Recommendations

(U) Lesson 2 - Variables Exercises

(U) Identify the type of each of the following variables, and add the type after each variable in a comment.

```
a = 2999
b = 90.0
c = "145"
d = "\u0CA0_\u0CA0"
e = "True"
f = True
g = len("sample")
h = 100**30
i = 1 >= 1
j = 30%7
k = 30/7
l = b + 7
m = 128 << 1
n = bin(255)
o = [m,l,k,n]
p = len(o)
```

(U) What value is in variable my_var at the end of these assignments?
Add a comparison after the last statement in the form of my_val ==

Doc ID: 6689693

```
my_var = 99
my_var += 11
my_var = str(my_var)
my_var *= 2
my_var = len(my_var)
my_var *= 4
```

Lesson 03: Flow Control

0

Updated over 1 year ago by [redacted] in COMP 3321

3 2 855 381

python

(U) Python flow control with conditionals and loops (if, while, for, range, etc.).

Recommendations

(U) Introduction

(U) If you have ever programmed before, you know one of the core building blocks of algorithms is flow control. It tells your program what to do next based on the state it is currently in.

(U) Comparisons

(U) First, let's look at how to compare values. The comparison operators are `>`, `>=`, `<`, `<=`, `!=`, and `==`. When working with numbers, they do what you think: return `True` or `False` depending on whether the statement is true or false.

```
2 < 3
```

```
2 > 5
```

```
x = 5
```

```
x == 6
```

```
x != 6
```

(U) Python 2.x will let you try to compare any two objects, no matter how different. The results may not be what you expect. Python 3.x only compares types where a comparison operation has been defined.

```
'apple' > 'orange'    # case-sensitive alphabetical
```

```
'apple' > 'Orange'
```

```
'apple' > ['orange']
```

```
'apple' > ('orange',)
```

(U) We will leave more discussion of comparisons for later, including how to intelligently compare objects that you create.

(U) Exercises

1. Write a `you_won` function that randomly picks a number from your price list (9.42, 5.67, 3.25, 13.40, and 7.50) and prints `True` or `False` depending on whether the random number is greater than 10.
2. Write a function `snack_check` that takes a string `snack` and returns `True` or `False` depending on whether or not it is your favorite snack.

(U) Conditional Execution: The `if` Statement

(U) The **if** statement is an important and useful tool. It basically says, "If a condition is true, do the requested operations."

```
def even(n):
    if (n % 2 == 0):
        print('I am even!')
```

```
even(2)
```

```
even(3)
```

```
even('hello')         # That was silly
```

(U) What if we want to be able to say we are not even? Or the user submitted a bad type? We use `else` and `elif` clauses.

```
def even(n):
    if (type(n) != int):
        print('I only talk about integers')
    elif (n % 2 == 0):
        print('I am even!')
    else:
        print('I am odd!')
```

```
even(2)
```

```
even(3)
```

```
even('hello')
```

(U) Exercises

1. Re-write the `snack_check` to take a string `snack` and prints an apporpriate response depending on whether the input is your favorite snack or not.
2. Write an `in_grocery_list` function that takes in a `grocery_item` prints a different message depending on whether `grocery_item` is in your grocery list.
3. Modify `in_grocery_list` to test if `grocery_item` is a string. Print a message warning the user if it is not.
4. Challenge: Re-write the `you_won` function to randomly choose a number from your price list and print appropriate message depending on whether you won (the number was greater than 10) or not. Also include the amount of change you will be receiving in your message. (Recall you are winning the amount change you would have owed...).
5. Advanced challenge: Write a function that imports `datetime` and uses it to determine the current time. This function should print an appropriate message based on the time ex: if the current time is between 0900 and 1000, print the message "Morning Lecture time!"

(U) Looping Behavior

(U) The `while` Loop

(U) The **while** is used for repeated operations that continue as long as an expression is true.

(U) The famous infinite loop:

```
while (2 + 2 == 4):
    print('forever')
```

(U) A mistake that may lead to an infinite loop:

```
i = 0

while (i <= 20):
    print(i)
```

(U) The below is probably a more sensible thing to type.

```
i = 0

while (i <= 20):
    print(i)
    i += 1
```

(U) break and continue

(U) For more control, we can use **break** and **continue** (they work just as in C). The **break** command will break out of the smallest `while` or `for` loop:

```
i = 0

while(True):
    i += 1
    print(i)
    if (i == 20):
        break
```

(U) The **continue** command will halt the current iteration of the loop and continue to the next value.

```
i = 0

while(True):
    i += 1
    if (i == 10):
        print("I am 10!")
        continue
    print(i)
    if (i == 20):
        break
```

(U) The `else` clause

(U) You can also have an `else` statement at the end of a loop. It will be run only if the loop completes normally, that is, when the conditional expression results in `False`. A `break` will skip it.

```
i = 0
while (i < 2):
    print(i)
    i += 1
else:
    print("This executes after the condition becomes false.")
print("Done!")
```

```
i = 0
while (i < 2):
    print(i)
    if True:
        break
    i += 1
else:
    print("This won't print because the loop was exited early.")
print("Done!")
```

(U) Exercises

Hint: you will not need `continue` or `break` for these exercises.

1. Previously we printed out many copies of a string using the `*` operator. Use a `while` loop to print out 10 copies of your favorite snack. Each copy can be on it's own line, that's fine.
2. Mix and match! Write a `while` loop that uses the `*` to print multiple copies of your favorite snack per line. Print out 10 lines with the number of copies per line corresponding to the line number (your first line will have one copy and your last line will have 10).
3. Challenge: Write a `while` loop that prints 100 copies of your favorite snack on one single (wrapped) line. Hint: use `+`.

(U) The `for` loop

(U) The `for` loop is probably the most used control flow element as it has the most functionality. It basically says, "for the following explicit items, do something." We are going to use the **list**type here. More interesting properties of this type will follow in another lesson.

```
for i in [1,2,3,4,5,'a','b','c']:
    print(i)
```

(U) The variable `i` "becomes" each value of the list and then the following code is executed:

```
for i in [1,2,3,4,5,'a','b','c']:
    print(i, type(i))
```

```
for c in 'orange':
    print(c)
```

(U) Exercises

1. Write a `for` loop that prints out each character in the string "blood-oxygenation level dependent functional magnetic resonance imaging" (Fun fact: this string is the longest entry in WordNet3.1 Index).
2. Take your grocery list of five items (or create one). Write a `for` loop to print out the message "Note to self, buy: " and then the grocery item.
3. Write a `for` loop that prints out a numbered list of your grocery items.
4. Clearly your favorite snack is more important than the other items on your list. Modify your `for` loop from Exercise 3 to use `break` stop printing once you have found your favorite snack in your list. Question: Could you have achieved the same result without using a `break`? Bonus: if your snack isn't in the list, have your code print a warning at the end.
5. Challenge: use the string method `split` to write a `for` loop that prints out each word in the string "blood-oxygenation level dependent functional magnetic resonance imaging". Hint: run `help(str.split)`

(U) `for` Loop Fodder: `range` and `xrange`

(U) Ok, that is great...but I want to print 1,000,000 numbers! The `range` function returns a list of values based on the arguments you provide. This is a simple way to generate 0 through 9:

```
print(range(10))
```

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

```
for i in range(10, 20):
    print(i)
```

```
for i in range(10, 20, 2):
    print(i)
```

```
for i in range(100, 0, -5):
    print(i)
```

(U) This makes a great tool for keeping a notion of the index of the loop!

```
a = "mystring"

for i in range(len(a)):
    print("The character at position " + str(i) + " is " +  a[i])
```

(U) Incidentally, the `enumerate` function is the preferred way of keeping track of the loop index:

```
for (i, j) in enumerate(a):
    print("The character at position " + str(i) + " is " +  j)
```

(U) In Python 3, the `range` function produces an **iterator**. For now, think of an **iterator** as an object that knows where to start, where to stop, and how to get from start to stop, but doesn't keep track of every step along the way all at once. We'll discuss iterators more later.

(U) In Python 2, `xrange` acts like Python 3's `range`. `range` in Python 2 produces a list, so the entire range is allocated in memory. You should almost always use `xrange` instead of `range` in Python 2.

```
b = range(100000000)   # Ohh, that was fast
```

```
b                      # It's just an object!
```

```
for i in range(10000):
    if (i % 2 == 0):
        print(i)
```

```
b = range(0, 1000000, 100)
```

```
b
```

```
b[0]
```

```
b[1]
```

```
b[2]
```

```
b[-1]
```

(U) Exercises

1. Use `range` to write a `for` loop to print out a numbered grocery list.
2. Use `enumerate` to print out a numbered grocery list. You've now done this three ways. What are some pros and cons to each technique? There are often several different ways to get the same output! However, usually one is more elegant than the others.
3. Use `range` to write a `for` loop that prints out 10 copies of your favorite snack. How does this compare to useing a `while` loop?
4. Challenge: Write a "Guess my number" game that generates a random number and gives your user a fixed number of guesses. Use `input` to get the user's guesses. Think about what loop type you might use and how you might provide feedback based on the user's guesses. Hint: what type does `input` return? You might need to convert this to a more useful type... However, now what happens if your user inputs something that isn't a number?

Doc ID: 6689693

Lesson 3 - Flow Control Exercises

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

0

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

3 331 157

python exercises

(U) Flow Control Exercises for COMP3321 Lesson 3

Recommendations

(U) Lesson 3 - Flow Control Exercises

(U) Change the loop below so that it prints numbers from 1 to 10.

```
for i in range(9):
    print(i)
```

(U) Using a for loop and enumerate, write a function `getindex(string, character)` to recreate the string method .index

```
"skyscraper".index('c')
# 4


getindex("skyscraper",'c')
# 4
```

(U) Using the shout function from the first set of basic exercises, write a shout_words(sentence) function that takes a string argument and "shouts" each word on its own line.

```
shout_words("Everybody likes bananas")
# EVERYBODY!
# LIKES!
# BANANAS!
```

Doc ID: 6689693

(U) Write an `extract_longer(length,sentence` function that takes a sentence and word length, then returns a list of the sentence's words that exceed the given length. If no words match the length, return False.

```
extract_longer(5, "Try not to interrupt the speaker.")
# ['interrupt','speaker.']

extract_longer(7, "Sorry about the mess.")
# False
```

Lesson 04: Container Data Types

0

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

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

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

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:

Doc ID: 6689693

```
[[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.
```

P.L. 86-36

```
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
```

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)[[redacted] ~]$ pip install ipython
  Downloading ipython-1.1.0.tar.gz (8.7MB): 8.7MB downloaded
...
Successfully installed ipython
Cleaning up...
(NEWENV)[[redacted] ~]$ 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)[[redacted] ~]$ pip install pyzmq tornado jinja2 pygments
(NEWENV)[[redacted] ~]$ ipython notebook --no-mathjax
```

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

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

Lesson 07: Object Orienteering: Using Classes

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!)

Doc ID: 6689693

(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.

Lesson 07: Supplement

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...