

Practical Python Programming

David Beazley (@dabeaz)

<http://www.dabeaz.com>

<http://www.dabeaz.com/python/practicalpython.zip>

Copyright (C) 2007-2017
All Rights Reserved

Table of Contents

- 0. Course Setup
- 1. Introduction
- 2. Working with Data
- 3. Program Organization
- 4. Classes
- 5. Inside the Python Object Model
- 6. Iterators and Generators
- 7. Some Advanced Topics
- 8. Testing and Debugging
- 9. Packages

This course is an introduction to Python primarily designed for programmers, scientists, or engineers who already know another programming language, but are new to Python. Various features of Python are introduced with a focus on using Python for various tasks involving scripting, data analysis, and file processing. By the end of this course, you will have an overview of most of Python's core features and know how to write short programs.

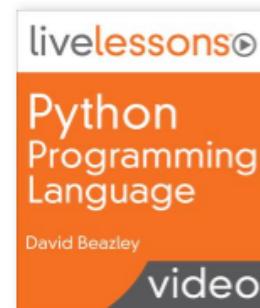
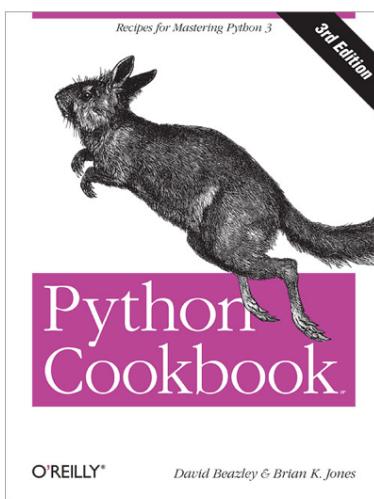
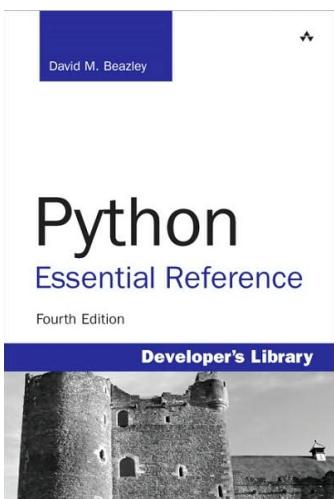
Take a Course

Go beyond the notes and self-study! Come to Chicago for a week and take this course in-person with Dave and a small group of enthusiastic Python learners. You'll cover everything here and more with interesting group discussion, extra projects, and insights that go far beyond the notes.

<http://www.dabeaz.com/chicago/index.html>



Books/Video



Python Programming Language

★★★★★ 50 reviews

by David Beazley

Publisher: Addison-Wesley Professional

Release Date: August 2016

ISBN: 9780134217314

Topics: Python

<https://www.safaribooksonline.com>

About the Course

- A comprehensive survey (what is Python all about and how does it work?).
- Using Python to do useful things
- Features of Python that distinguish it from other programming languages

Prerequisites

- This class involves programming
- You should know how to use your computer (files, folders, network, editors, shells, etc.)
- You should know how to program in some other programming language (C++, Java, Perl, etc.)
- You should be familiar with basic concepts of programming (looping, arrays, etc.)

System Requirements

- This course describes Python 3.6
- In order to work exercises, you need to download and install the following

<http://www.dabeaz.com/python/practicalpython.zip>

- Extract the zip file someplace where it's easy to locate (i.e., C:\practical-python)
- Look at the "Course Setup" section of notes

Presentation Format

- Fast-paced discussion with hands-on exercises
- Please ask questions
- Don't hesitate to ask for more detail
(especially if something is broken)
- Please work together

Course Exercises

- The pace of this course is rapid
- Some exercises are harder than others
- You may not be able to create your own solution from scratch to every problem
- Solutions are always provided
- Copy that code if needed

Note on Frameworks

- Python has a lot of large application frameworks
 - Scientific computing
 - Web programming
 - Graphical User Interfaces
- This class is about the core Python language, standard library, and programming idioms
- Frameworks require their own course

Section 0

Course Setup

Required Files

- Course requires Python 3.6+
- Where to get Python (if not installed)

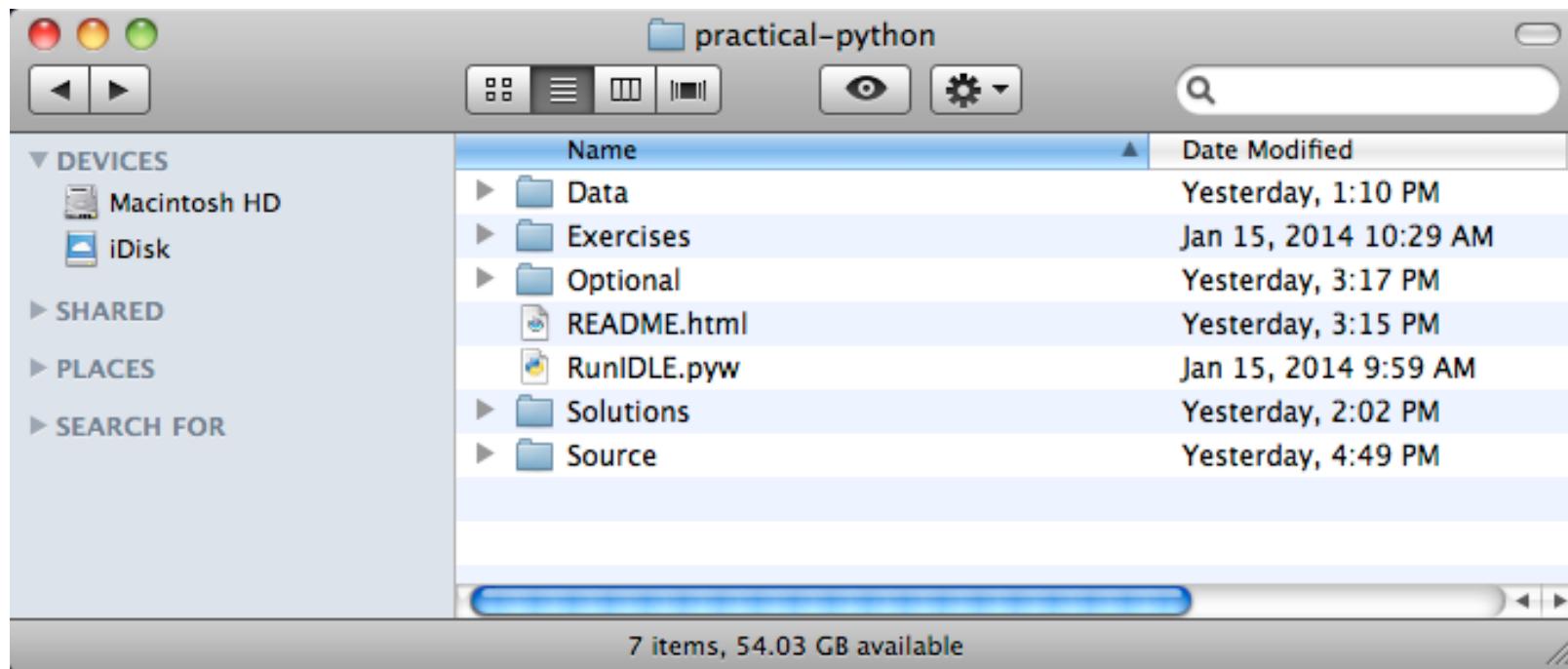
<http://www.python.org>

- Exercises for this class

<http://www.dabeaz.com/python/practicalpython.zip>

Setting up Your Environment

- Extract practical-python on your machine



- This folder is where you will do your work

Class Exercises

- Exercise descriptions are found in
[practical-python/Exercises/index.html](#)
- All exercises have solution code

One nice thing about the `csv` module is that it deals with a variety of output, you'll notice that it has stripped the double-quotes away from the output. Modify your `pconst.py` program so that it uses the `csv` module for output.



Look for the link at the bottom!

Solution Code

- Fully working solution code is found in
[practical-python/Solutions/](#)
- It's okay to look at it, copy it, etc.
- However, I encourage you to try and come up with your own solution first.

General Tips

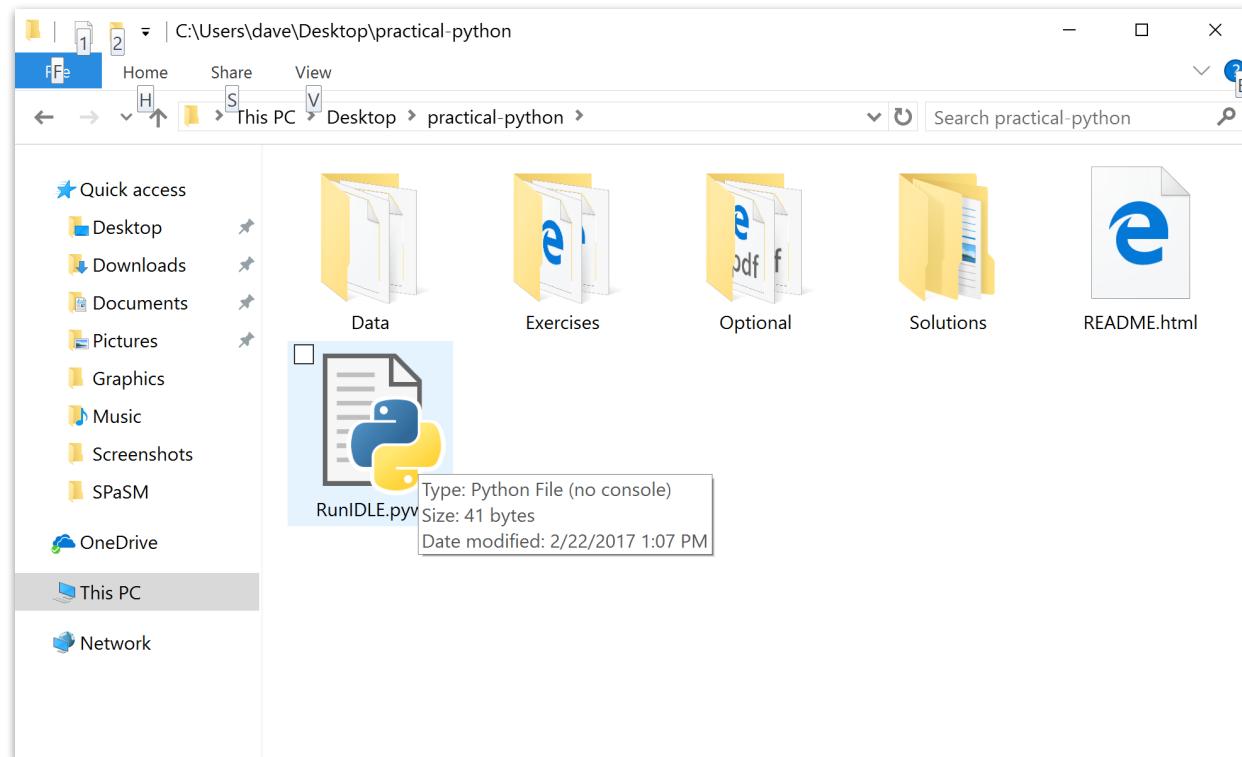
- We will be writing a lot of programs that access data files in practical-python/Data
- Save your programs in the "practical-python/" directory so that the names of these files are easy to access.
- Some exercises are more difficult than others. Please copy code from the solution and study it if necessary.

Text Editors

- You may use any text editor that you are familiar with for this course (e.g., vim, emacs, etc.)
- Python also comes with a simple development environment called IDLE.
- It's not the most advanced tool, but it works
- Follow the instructions on the next two slides

Running IDLE (Windows)

- Find RunIDLE in the practical-python/ folder
- Double-click to start the IDLE environment



Running IDLE (Mac/Unix)

- Go into the practical-python/ directory
- Type the following command in a command shell

```
bash % python3 -m idlelib.idle
```

- Note: Typing 'idle' at the shell might also work.

Section I

Introduction to Python

What is Python?

- An interpreted high-level programming language.
- Similar to Perl, Ruby, Tcl, and other so-called "scripting languages."
- Created by Guido van Rossum around 1990.
- Named in honor of Monty Python

Where to Get Python?

<http://www.python.org>

- Downloads
- Documentation and tutorial
- Community Links
- Third party packages
- News and more

Why was Python Created?

"My original motivation for creating Python was the perceived need for a higher level language in the Amoeba [Operating Systems] project. I realized that the development of system administration utilities in C was taking too long. Moreover, doing these things in the Bourne shell wouldn't work for a variety of reasons. ... So, there was a need for a language that would bridge the gap between C and the shell."

- Guido van Rossum

Running Python

- Python programs run inside an interpreter
- The interpreter is a simple "console-based" application that normally starts from a command shell (e.g., the Unix shell)

```
bash % python3
Python 3.6.0 (default, Jan  3 2017, 10:48:19)
[GCC 4.2.1 Compatible Apple LLVM 6.1.0 (clang-602.0.53)] or
Type "help", "copyright", "credits" or "license" for more i
>>>
```

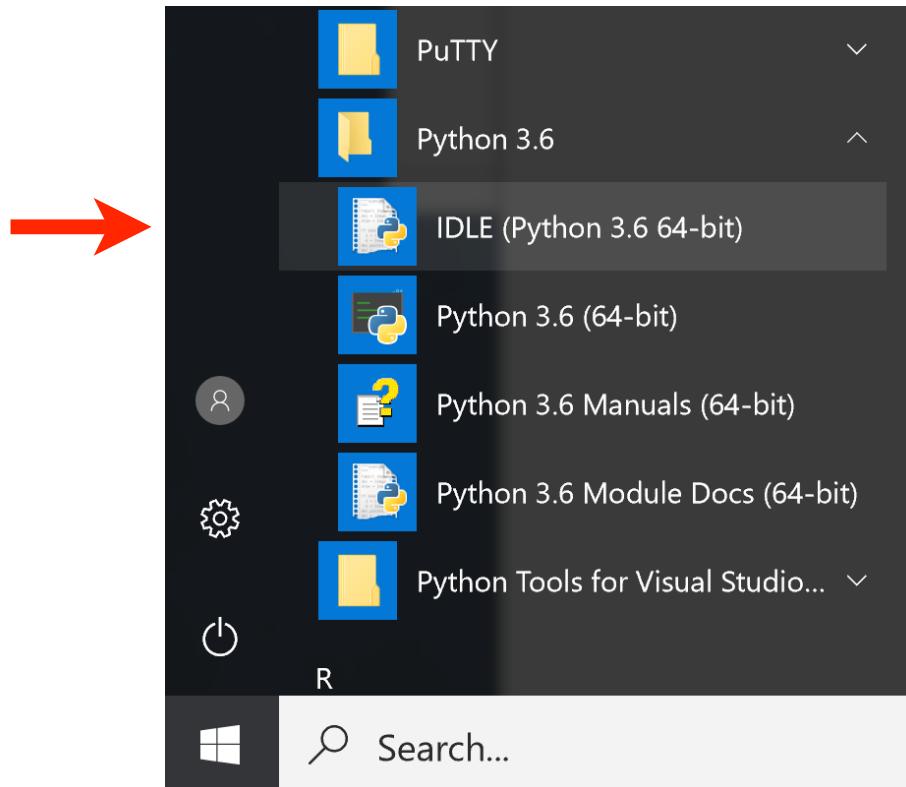
- Expert programmers usually have no problem using the interpreter in this way, but it's not so user-friendly for beginners

IDLE

- Python includes a simple integrated development called IDLE (which is another Monty Python reference)
- It's not the most sophisticated environment but it's already installed and it works
- Most working Python programmers tend to use something else, but it is fine for this class.

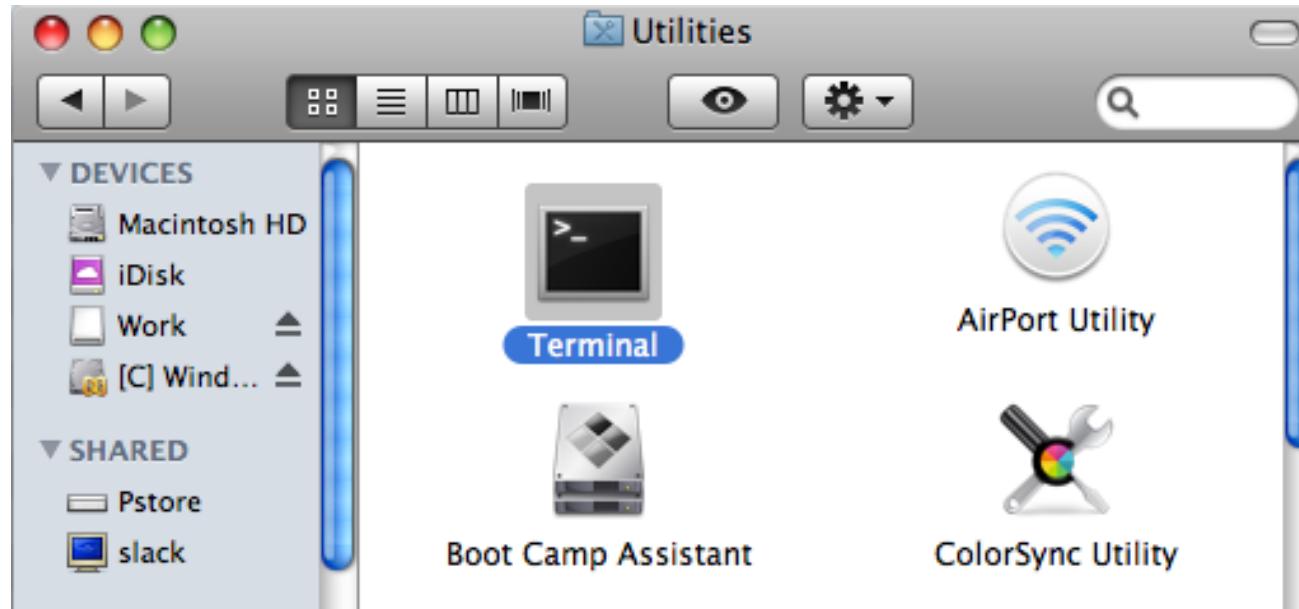
IDLE on Windows

- Look for it in the launcher



IDLE on other Systems

- Launch a terminal or command shell



- Type the following command to launch IDLE

```
bash % python3 -m idlelib.idle
```

The Python Interpreter

- When you start Python, you get an "interactive" mode where you can experiment
- If you start typing statements, they will run immediately
- No edit/compile/run/debug cycle

Interactive Mode

- The interpreter runs a "read-eval" loop

```
>>> print('hello world')
hello world
>>> 37*42
1554
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>>
```

- Executes simple statements typed in directly
- Very useful for debugging, exploration

Interactive Mode

- Some notes on using the interactive shell

`>>>` is the interpreter
prompt for starting a
new statement

... is the interpreter
prompt for continuing
a statement (it may be
blank in some tools)

```
>>> print('hello world')
hello world
>>> 37*42
1554
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>>
```

Enter a blank line to
finish typing and to run

Interactive Mode

- Use underscore (`_`) for the last result

```
>>> 37*42  
1554  
>>> _ * 2  
3108  
>>> _ + 50  
3158  
>>>
```

- Note: This only works in interactive mode (you never use `_` in a program)

Getting Help

- `help(name)` command

```
>>> help(range)
Help on class range in module builtins:

class range(object)
    range(stop) -> range object
    range(start, stop[, step]) -> range object

    Return an object that produces a sequence of integers from start (inclusive)
    to stop (exclusive) by step. range(i, j) produces i, i+1, i+2, ..., j-1.
    start defaults to 0, and stop is omitted! range(4) produces 0, 1, 2, 3.
    These are exactly the valid indices for a list of 4 elements.
    When step is given, it specifies the increment (or decrement).
>>>
```

- Type `help()` with no name for interactive help
- Documentation at <http://docs.python.org>

Exercise I.I

Time: 10 minutes

Creating Programs

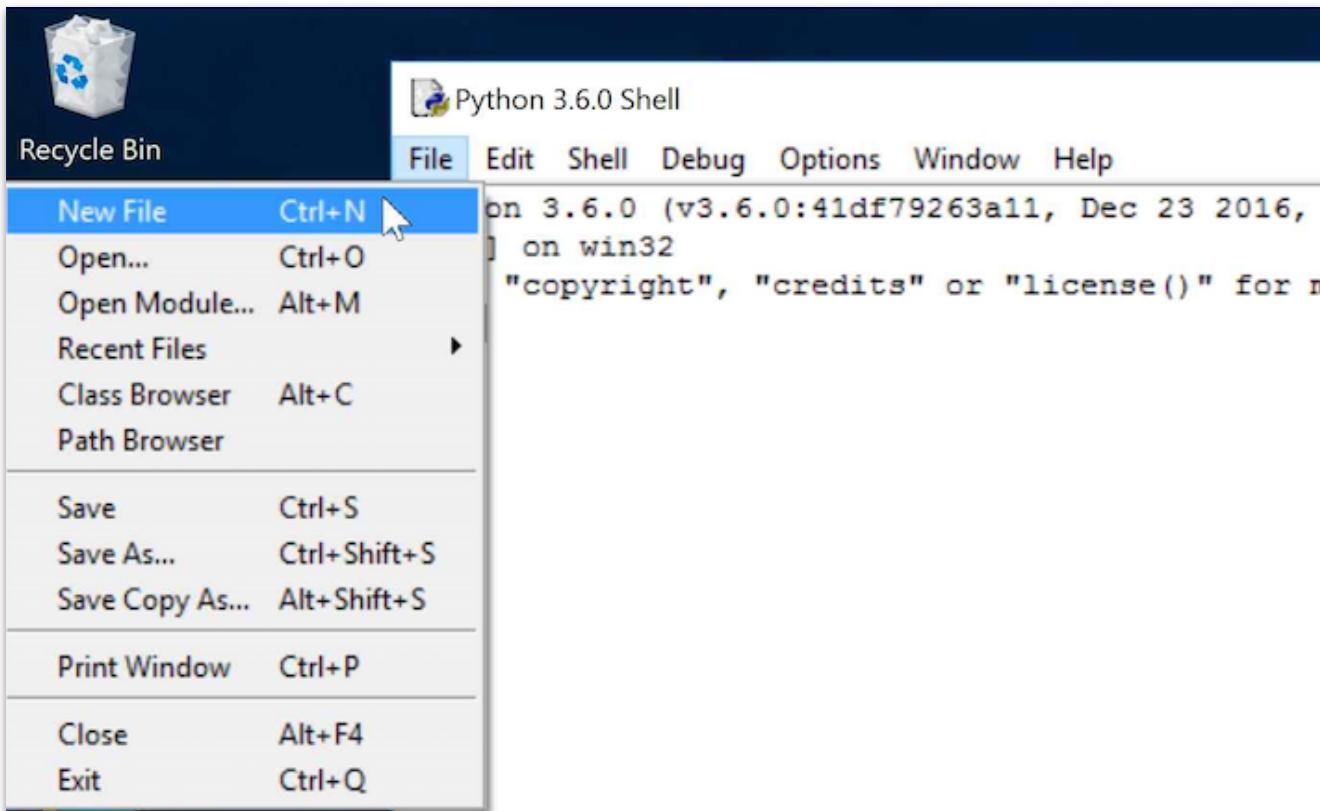
- Programs are put in .py files

```
# helloworld.py  
print('hello world')
```

- Create with your favorite editor (e.g., emacs)
- Can also edit programs with IDLE or other Python IDE (too many to list)

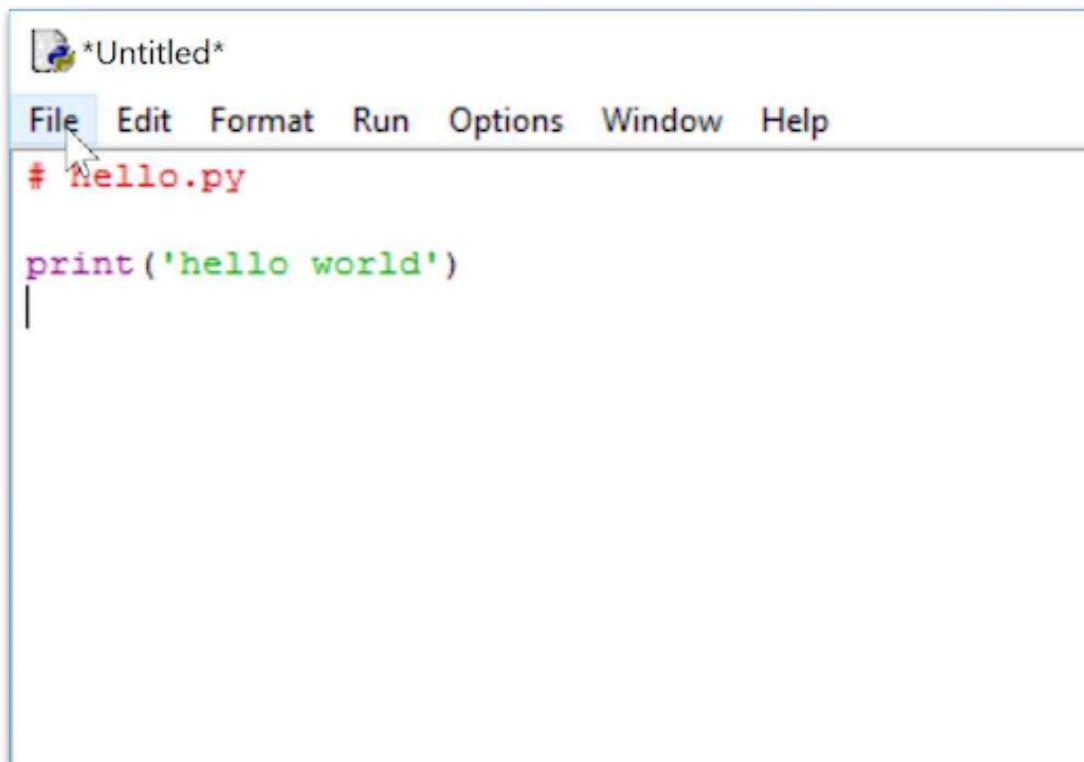
Creating Programs

- Creating a new program in IDLE



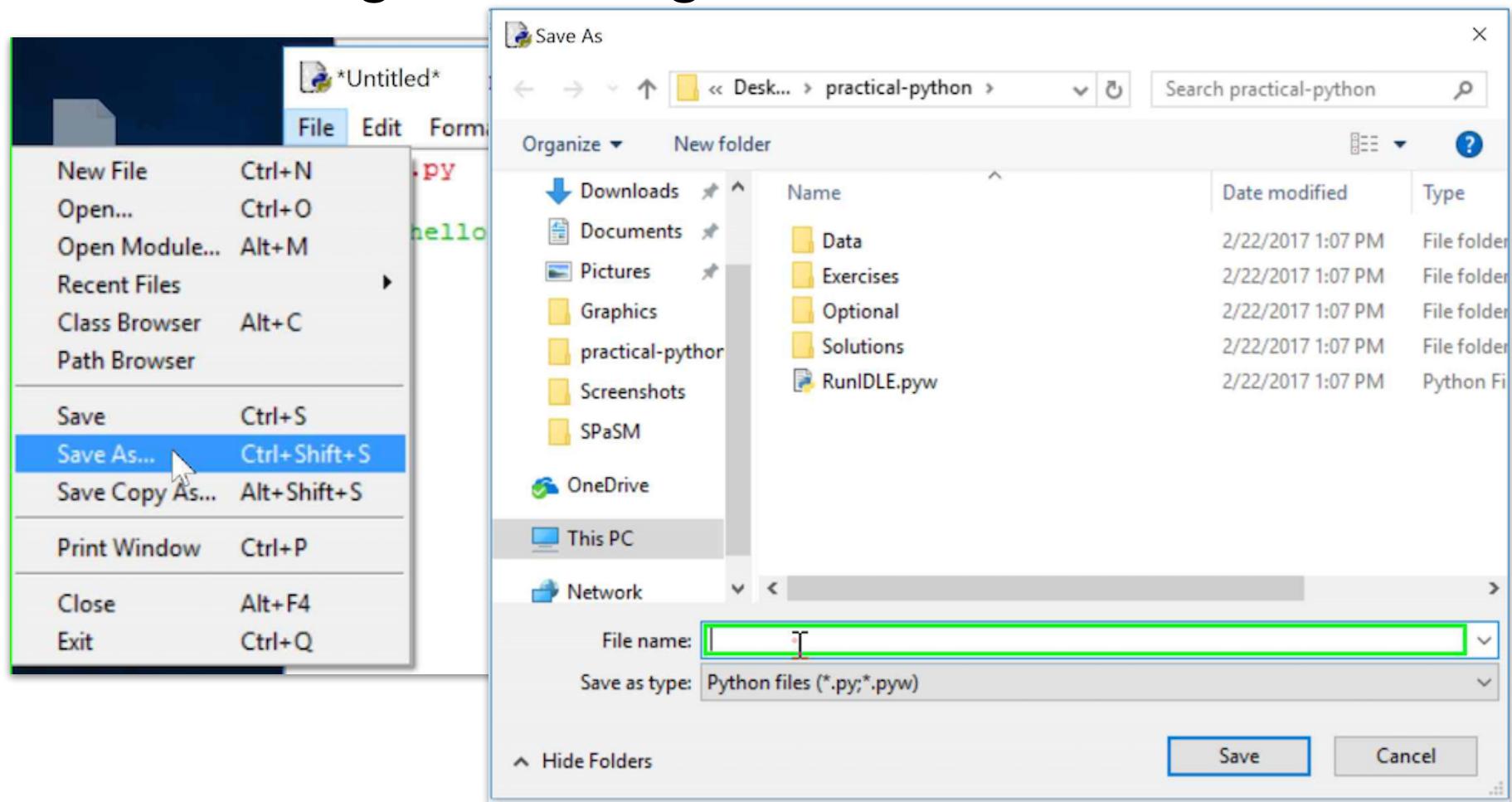
Creating Programs

- Editing a new program in IDLE



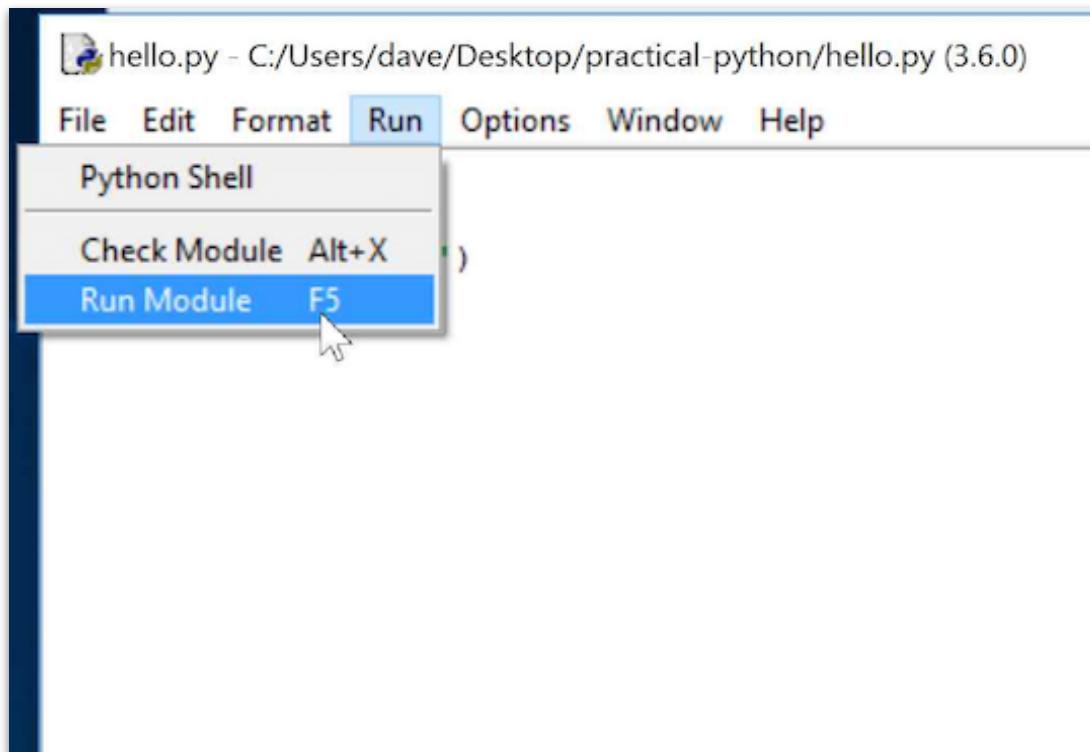
Creating Programs

- Saving a new Program in IDLE



Running Programs (IDLE)

- Select "Run Module" (F5)



- Will see output in IDLE shell window

Running Programs

- In production environments, Python may be run from command line or a script
- Command line (Unix)

```
bash % python3 hello.py
hello world
bash %
```

- Command shell (Windows)

```
C:\SomeFolder>hello.py
hello world
```

```
C:\SomeFolder>c:\python36\python hello.py
hello world
```

A Sample Program

- The Sears Tower Problem

One morning, you go out and place a dollar bill on the sidewalk by the Sears tower. Each day thereafter, you go out double the number of bills. How long does it take for the stack of bills to exceed the height of the tower?



A Sample Program

```
# sears.py

bill_thickness = 0.11 * 0.001      # Meters (0.11 mm)
sears_height = 442                  # Height (meters)
num_bills = 1
day = 1

while num_bills * bill_thickness < sears_height:
    print(day, num_bills, num_bills * bill_thickness)
    day = day + 1
    num_bills = num_bills * 2

print('Number of days', day)
print('Number of bills', num_bills)
print('Final height', num_bills * bill_thickness)
```

A Sample Program

- Output

```
bash % python3 sears.py
1 1 0.00011
2 2 0.00022
3 4 0.00044
4 8 0.00088
5 16 0.00176
6 32 0.00352
...
21 1048576 115.34336
22 2097152 230.68672
Number of days 23
Number of bills 4194304
Final height 461.37344
```

Python 101 : Statements

- A Python program is a sequence of statements
- Each statement is terminated by a newline
- Statements are executed one after the other until you reach the end of the file.

Python 101 : Comments

- Comments are denoted by #

```
# This is a comment  
height      = 442                      # Meters
```

- Extend to the end of the line

Python 101: Variables

- A variable is a name for a value
- Variable names follow same rules as C/
Java [A-Za-z_][A-Za-z0-9_]*
- You do not declare types (int, float, etc.)

```
height = 442                      # An integer
height = 442.0                     # Floating point
height = 'Really tall'             # A string
```

- Types are associated with the values

Python 101: Case Sensitivity

- Python is case sensitive
- These are all different variables:

```
name = 'Jake'  
Name = 'Elwood'  
NAME = 'Guido'
```

- Language statements are always lower-case

```
while x < 0:                  # OK  
WHILE x < 0:                  # ERROR
```

- So, no shouting please...

Python 101: Looping

- The `while` statement executes a loop

```
while num_bills * bill_thickness < sears_height:  
    print(day, num_bills, num_bills * bill_thickness)  
    day = day + 1  
    num_bills = num_bills * 2  
  
print('Number of days', day)
```

repeated statements

- Executes the indented statements underneath while the condition is true

Python 101 : Indentation

- Indentation must be consistent

```
while num_bills * bill_thickness < sears_height:  
    print(day, num_bills, num_bills * bill_thickness)  
    day = day + 1 ←  
    num_bills = num_bills * 2 → (error)
```

- Colon (:) indicates the start of a block

```
while num_bills * bill_thickness < sears_height:  
    ↑
```

Python 101 : Indentation

- There is a preferred indentation style
 - Always use spaces
 - Use 4 spaces per level
 - Avoid tabs (convert to spaces)
- Always use a Python-aware editor

Python 101 : Conditionals

- If-else

```
if a < b:  
    print('Computer says no')  
else:  
    print('Computer says yes')
```

- If-elif-else

```
if a < b:  
    print('Computer says no')  
elif a > b:  
    print('Computer says yes')  
else:  
    print('Computer says maybe')
```

Python 101 : Printing

- The print function

```
print(x)
print(x, y, z)
print('Your name is', name)
print(x, end=' ') # Omits newline
```

- Produces a single line of text
- Items are separated by spaces
- Always prints a newline unless optional end argument is supplied

Python 101 : User Input

- To read a line of typed user-input

```
name = input('Enter your name: ')
```

- Prints a prompt, returns the typed response
- This might be useful for small programs or for simple debugging
- It is not widely used for real programs (we're rarely going to use it in this class)

Python 101 : pass statement

- Sometimes you will need to specify an empty block of code (like {} in C/Java)

```
if name in namelist:  
    # Not implemented yet (or nothing)  
    pass  
else:  
    statements
```

- **pass** is a "no-op" statement
- It does nothing, but serves as a placeholder for statements (possibly to be added later)

Exercise I.2

Time: 10 minutes

Numbers

- Python has 4 types of numbers
 - Booleans
 - Integers
 - Floating point
 - Complex (imaginary numbers)

Booleans

- Two values: True, False

```
a = True  
b = False
```

- Evaluated as integers with value 1,0

```
c = 4 + True      # c = 5  
d = False  
if d == 0:  
    print('d is False')
```

- Although doing that in practice would be odd

Integers

- Signed values of arbitrary size

```
a = 37
b = -299392993727716627377128481812241231
c = 0x7fa8          # Hexadecimal
d = 0o253          # Octal
e = 0b10001111    # Binary
```

- Common operations

+	Add	<<	Bit shift left
-	Subtract	>>	Bit shift right
*	Multiply	&	Bit-wise AND
/	Divide (a float)		Bit-wise OR
//	Floor divide	^	Bit-wise XOR
%	Modulo	~	Bit-wise NOT
**	Power	abs(x)	Absolute value

Comparisons

- Comparison relational operators

< > <= >= == !=

- Boolean expressions (and, or, not)

```
if b >= a and b <= c:  
    print('b is between a and c')  
  
if not (b < a or b > c):  
    print('b is still between a and c')
```

Floating point (float)

- Use a decimal or exponential notation

a = 37.45

b = 4e5

c = -1.345e-10

- Represented as double precision using the native CPU representation (IEEE 754)

17 digits of precision

Exponent from -308 to 308

- Same as the C double type

Floating point

- Be aware that floating point numbers are inexact when representing decimal

```
>>> a = 2.1 + 4.2
>>> a == 6.3
False
>>> a
6.30000000000001
>>>
```

- This is not Python, but the underlying floating point hardware on the CPU.
- The result of a calculation may not be quite what you expect (emphasis, not a Python bug)

Floating Point Operators

+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulo (remainder)
**	Power
abs(x)	Absolute value

- Additional functions are in the math module

```
import math
a = math.sqrt(x)
b = math.sin(x)
c = math.cos(x)
d = math.tan(x)
e = math.log(x)
```

Converting Numbers

- Type name can be used to convert

```
a = int(x)          # Convert x to integer  
b = float(x)        # Convert x to float
```

- Example:

```
>>> a = 3.14159  
>>> int(a)  
3  
>>>
```

- Also work with strings containing numbers

```
>>> a = '3.14159'  
>>> float(a)  
3.14159  
>>>
```

Exercise I.3

Time: 15 minutes

Strings

- Literals written in programs with quotes

```
a = 'Yeah but no but yeah but...'
```

```
b = "computer says no"
```

```
c = '''
```

```
Look into my eyes, look into my eyes,  
the eyes, the eyes, the eyes,  
not around the eyes,  
don't look around the eyes,  
look into my eyes, you're under.  
'''
```

- Standard escape characters work (e.g., '\n')
- Triple quotes capture all text enclosed

String Escape Codes

- Standard escape codes work

'\n'	Line feed
'\r'	Carriage return
'\t'	Tab
'\''	Literal single quote
'\"'	Literal double quote
'\\'	Backslash

- The codes are inspired by C

String Representation

- Contain raw unicode code points



```
a = '\xf1'          # a = 'ñ'  
b = '\u2200'        # b = '∀'  
c = '\U0001D122'   # c = '♪'  
d = '\N{FOR ALL}'  # d = '∀'
```

String Representation

- Strings work like an array : $s[n]$

```
a = 'Hello world'  
b = a[0]      # b = 'H'  
c = a[4]      # c = 'o'  
d = a[-1]     # d = 'd'  (Taken from end of string)
```

- Slicing/substrings : $s[start:end]$

```
d = a[:5]      # d = 'Hello'  
e = a[6:]       # e = 'world'  
f = a[3:8]      # f = 'lo wo'  
g = a[-5:]      # g = 'world'
```

- Concatenation (+)

```
a = 'Hello' + 'World'  
b = 'Say ' + a
```

More String Operations

- Length (`len`)

```
>>> s = 'Hello'  
>>> len(s)  
5  
>>>
```

- Membership test (`in`, `not in`)

```
>>> 'e' in s  
True  
>>> 'x' in s  
False  
>>> 'hi' not in s  
True
```

- Replication (`s*n`)

```
>>> s = 'Hello'  
>>> s*5  
'HelloHelloHelloHelloHello'  
>>>
```

String Methods

- Strings have "methods" that perform various operations with the string data.
- Stripping any leading/trailing whitespace

```
t = s.strip()
```

- Case conversion

```
t = s.lower()  
t = s.upper()
```

- Replacing text

```
t = s.replace('Hello', 'Hallo')
```

More String Methods

```
s.endswith(suffix)      # Check if string ends with suffix
s.find(t)               # First occurrence of t in s
s.index(t)              # First occurrence of t in s
s.isalpha()              # Check if characters are alphabetic
s.isdigit()              # Check if characters are numeric
s.islower()              # Check if characters are lower-case
s.isupper()              # Check if characters are upper-case
s.join(slist)            # Joins lists using s as delimiter
s.lower()                # Convert to lower case
s.replace(old,new)       # Replace text
s.rfind(t)               # Search for t from end of string
s.rindex(t)              # Search for t from end of string
s.split([delim])        # Split string into list of substrings
s.startswith(prefix)     # Check if string starts with prefix
s.strip()                # Strip leading/trailing space
s.upper()                # Convert to upper case
```

- Consult a reference for gory details

String Mutability

- Strings are "immutable" (read only)
- Once created, the value can't be changed

```
>>> s = 'Hello World'  
>>> s[1] = 'a'  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'str' object does not support item assignment  
>>>
```

- All operations and methods that manipulate string data always create new strings

String Conversions

- Use `str()` to convert a value to a string

```
>>> x = 42  
>>> str(x)  
'42'  
>>>
```

- Resulting text is same as produced by `print`

Byte Strings

- A string of 8-bit bytes

```
data = b'Hello World\r\n'
```

- Most of the usual string operations work

```
len(data)                      # -> 13
data[0:5]                       # -> b'Hello'
data.replace(b'Hello', b'Cruel') # -> b'Cruel World\r\n'
```

- Indexing is a bit different (returns byte values)

```
data[0]    # -> 72 (ASCII code for 'H')
```

- Conversion to/from text

```
text = data.decode('utf-8') # bytes -> text
data = text.encode('utf-8') # text -> bytes
```

Raw Strings

- String with uninterpreted backslash

```
r'c:\newdata\test'    # Raw (uninterpreted backslash)
```

- String is the literal text exactly as typed
- Sometimes used in situations where the backslash (\) has special significance
- Example: filenames, regular expressions, etc.

f-Strings

- String with formatted expression substitution

```
name = 'IBM'  
shares = 100  
price = 91.1
```

```
>>> a = f'{name:>10s} {shares:10d} {price:10.2f}'  
>>> a  
'           IBM          100      91.10'
```

```
>>> b = f'Cost = ${shares*price:0.2f}'  
>>> b  
'Cost = $9110.00'  
>>>
```

- Note: Requires Python 3.6 or newer

Exercise I.4

Time: 10 minutes

String Splitting

- Strings often represent fields of data
- To work with each field, split into a list

```
>>> line = 'GOOG,100,490.10'  
>>> row = line.split(',')  
>>> row  
['GOOG', '100', '490.10']  
>>>
```

- Example: When reading data from a file, you might read each line and then split the line into columns.

Lists

- A array of arbitrary values

```
names = [ 'Elwood', 'Jake', 'Curtis' ]  
nums  = [ 39, 38, 42, 65, 111]
```

- Adding new items (append, insert)

```
names.append('Murphy')      # Adds at end  
names.insert(2, 'Aretha')   # Inserts in middle
```

- Concatenation : s + t

```
s = [1, 2, 3]  
t = ['a', 'b']
```

s + t → [1, 2, 3, 'a', 'b']

Lists (cont)

- Lists are indexed by integers (starting at 0)

```
names = [ 'Elwood', 'Jake', 'Curtis' ]
```

```
names[0] -----> 'Elwood'  
names[1] -----> 'Jake'  
names[2] -----> 'Curtis'
```

- Negative indices are from the end

```
names[-1] -----> 'Curtis'
```

- Changing one of the items

```
names[1] = 'Joliet Jake'
```

More List Operations

- Length (`len`)

```
>>> names = ['Elwood', 'Jake', 'Curtis']
>>> len(names)
3
>>>
```

- Membership test (`in`, `not in`)

```
>>> 'Elwood' in names
True
>>> 'Britney' not in names
True
>>>
```

- Replication (`s * n`)

```
>>> s = [1, 2, 3]
>>> s * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>>
```

List Iteration & Search

- Iterating over the list contents

```
for name in names:  
    # use name  
    ...
```

- Similar to a 'foreach' statement from other programming languages
- To find something quickly, use `.index()`

```
>>> names = ['Elwood', 'Jake', 'Curtis']  
>>> names.index('Curtis')  
2  
>>> names[2]  
'Curtis'  
>>>
```

List Removal

- Removing an item

```
names.remove('Curtis')
```

- Deleting an item by index

```
del names[2]
```

- Removal results in items moving down to fill the space vacated (i.e., no "holes").

List Sorting

- Lists can be sorted "in-place" (sort method)

```
s = [10, 1, 7, 3]
s.sort()                      # s = [1, 3, 7, 10]
```

- Sorting in reverse order

```
s = [10, 1, 7, 3]
s.sort(reverse=True)  # s = [10, 7, 3, 1]
```

- Sorting works with any ordered data

```
s = ['foo', 'bar', 'spam']
s.sort()                      # s = ['bar', 'foo', 'spam']
```

Lists and Math

- Caution : Lists weren't designed for "math"

```
>>> nums = [1, 2, 3, 4, 5]
>>> nums * 2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>>> nums + [10, 11, 12, 13, 14]
[1, 2, 3, 4, 5, 10, 11, 12, 13, 14]
>>>
```

- They don't represent vectors/matrices
- Not the same as in MATLAB, Octave, IDL, etc.
- There are some add-ons for this (e.g., numpy)

Exercise 1.5

Time: 10 minutes

File Input and Output

- Opening a file

```
f = open('foo.txt', 'rt')      # Open for reading (text)  
g = open('bar.txt', 'wt')      # Open for writing (text)
```

- To read data

```
data = f.read([maxbytes])    # Read up to maxbytes bytes
```

- To write text to a file

```
g.write('some text')
```

- To close when you're done

```
f.close()
```

File Management

- Files should be properly closed when

```
f = open(filename, 'rt')
# Use the file f
...
f.close()
```

- Preferred approach: use the 'with' statement

```
with open(filename, 'rt') as f:
    # Use the file f
    ...
file f → statements
closed here
```

- This automatically closes the file when control leaves the indented code block

Reading File Data

- Reading an entire file all at once as a string

```
with open(filename, 'rt') as f:  
    data = f.read()
```

- Reading an entire text-file line-by-line

```
with open(filename, 'rt') as f:  
    for line in f:  
        # Process the line  
        ...
```

Writing To a File

- Writing string data

```
with open('outfile', 'wt') as f:  
    f.write('Hello World\n')  
    ...
```

- Redirecting the print function

```
with open('outfile', 'wt') as f:  
    print('Hello World', file=f)  
    ...
```

Exercise I.6

Time: 15 minutes

Simple Functions

- Use functions for code you want to reuse

```
def sumcount(n):
    """
        Returns the sum of the first n integers
    """
    total = 0
    while n > 0:
        total += n
        n -= 1
    return total
```

- Calling a function

```
a = sumcount(100)
```

- A function is a series of statements that perform some task and return a result

Library Functions

- Python comes with a large standard library
- Library modules accessed using import

```
import math  
x = math.sqrt(10)
```

```
import urllib.request  
u = urllib.request.urlopen('http://www.python.org/')  
data = u.read()
```

- Will cover in more detail later

Exception Handling

- Errors are reported as exceptions
- An exception causes the program to stop

```
>>> int('N/A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'N/A'
>>>
```

- For debugging, message describes what happened, where the error occurred, along with a traceback.

Exceptions

- Exceptions can be caught and handled
- To catch, use try-except statement

```
for line in f:  
    fields = line.split()  
    try:  
        shares = int(fields[1])  
    except ValueError:  
        print("Couldn't parse", line)  
    ...
```

Name must match the kind of error
you're trying to catch

```
>>> int('N/A')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: invalid literal for int() with base 10: 'N/A'  
>>>
```

Exceptions

- To raise an exception, use the `raise` statement

```
raise RuntimeError('What a kerfuffle')
```

- Will cause the program to abort with an exception traceback (unless caught by `try-except`)

```
% python3 foo.py
Traceback (most recent call last):
  File "foo.py", line 21, in <module>
    raise RuntimeError("What a kerfuffle")
RuntimeError: What a kerfuffle
```

Summary

- This has been an overview of simple Python
- Enough to write basic programs
- Have to know the core datatypes and a few basics (loops, conditions, files, etc.)

Exercise 1.7

Time: 15 minutes

Section 2

Working with Data

Overview

- Most programs work with data
- In this section, we look at how Python programmers represent and work with data
- Common programming idioms
- How to (not) shoot yourself in the foot

Primitive Datatypes

- Python has a few primitive types of data
 - Integers
 - Floating point numbers
 - Strings (text)
- Obviously, all programs use these

None type

- Nothing, nil, null, nada

```
email_address = None
```

- This is often used as a placeholder for optional or missing value

```
if email_address:  
    send_email(email_address, msg)
```

Data Structures

- Real programs have more complex data
- Example: A holding of stock

100 shares of GOOG at \$490.10

- An "object" with three parts
 - Name ("GOOG", a string)
 - Number of shares (100, an integer)
 - Price (490.10, a float)

Tuples

- A collection of values grouped together
- Example:

```
s = ('GOOG', 100, 490.1)
```

- Sometimes the () are omitted in syntax

```
s = 'GOOG', 100, 490.1
```

- Special cases (0-tuple, 1-tuple)

```
t = ()                      # An empty tuple
w = ('GOOG',)                # A 1-item tuple
```

Tuple Use

- Tuples are usually used to represent simple records or structures

```
contact = ('David Beazley', 'dave@dabeaz.com')
stock   = ('GOOG', 100, 490.1)
host    = ('www.python.org', 80)
```

- A single "object" of multiple parts
- Analogy: A single row in a database table

Tuples (cont)

- Tuple contents are ordered (like an array)

```
s = ('GOOG', 100, 490.1)
name = s[0]      # 'GOOG'
shares = s[1]    # 100
price = s[2]     # 490.1
```

- However, the contents can't be modified

```
>>> s[1] = 75
TypeError: object does not support item assignment
```

- You can, however, make a new tuple

```
s = (s[0], 75, s[2])
```

Tuple Packing

- Tuples are focused more on packing related items together into a single "entity"

```
s = ('GOOG', 100, 490.1)
```

- The tuple is then easy to pass around to other parts of a program as a single object

Tuple Unpacking

- To use the tuple elsewhere, you can unpack its parts into variables

```
name, shares, price = s  
  
print('Cost', shares * price)
```

- Number of variables must match tuple structure

```
name, shares = s      # ERROR  
  
Traceback (most recent call last):  
...  
ValueError: too many values to unpack
```

Tuple Commentary

- Are tuples just a read-only list? No.
- Tuples are most often used for a single item consisting of multiple parts

```
record = ('GOOG', 100, 490.1)
```

- Lists are usually a collection of distinct items (typically all of the same type)

```
names = ['Elwood', 'Jake', 'Curtis']
```

Dictionaries

- A hash table or associative array
- A collection of values indexed by "keys"
- The keys serve as field names
- Example:

```
s = {  
    'name' : 'GOOG',  
    'shares' : 100,  
    'price' : 490.1  
}
```

Dictionaries

- Getting values: Use the key names

```
>>> print(s['name'], s['shares'])  
GOOG 100  
>>> s['price']  
490.10  
>>>
```

- Adding/modifying values :Assign to key names

```
>>> s['shares'] = 75  
>>> s['date'] = '6/6/2007'  
>>>
```

- Deleting a value

```
>>> del s['date']  
>>>
```

Dictionaries

- Dictionaries are useful when
 - there are many different values
 - The values will be modified/manipulated
 - You also get better code clarity

`s['date']` vs `s[4]`

Exercise 2.I

Time : 10 minutes

Containers

- Programs often have to work many objects
 - A portfolio of stocks
 - A table of stock prices
 - Three choices:
 - Lists (ordered data)
 - Dictionaries (unordered data)
 - Sets (unordered collection)

Lists as a Container

- Use a list when the order of data matters
- Lists can hold any kind of object
- Example: A list of tuples

```
portfolio = [  
    ('GOOG', 100, 490.1),  
    ('IBM', 50, 91.3),  
    ('CAT', 150, 83.44)  
]
```

portfolio[0] → ('GOOG', 100, 490.1)
portfolio[1] → ('IBM', 50, 91.3)

List Construction

- Example of building a list from scratch

```
records = []          # Initial empty list

# Use .append() to add more items
records.append(('GOOG', 100, 490.10))
records.append(('IBM', 50, 91.3))
...
...
```

- Example: Reading records from a file

```
records = []          # Initial empty list

with open('portfolio.csv', 'rt') as f:
    for line in f:
        row = line.split(',')
        records.append((row[0], int(row[1])), float(row[2]))
```

Dicts as a Container

- Dictionaries are useful if you want fast random lookups (by key name)
- Example: A dictionary of stock prices

```
prices = {  
    'GOOG' : 513.25,  
    'CAT'   : 87.22,  
    'IBM'   : 93.37,  
    'MSFT'  : 44.12  
    ...  
}  
  
>>> prices['IBM']  
93.37  
>>> prices['GOOG']  
513.25  
>>>
```

Dict Construction

- Example of building a dict from scratch

```
prices = {}      # Initial empty dict

# Insert new items
prices['GOOG'] = 513.25
prices['CAT'] = 87.22
prices['IBM'] = 93.37
```

- Example: Populating from a file

```
prices = {}      # Initial empty dict

with open('prices.csv', 'rt') as f:
    for line in f:
        row = line.split(',')
        prices[row[0]] = float(row[1])
```

Dictionary Lookups

- To test for existence of a key

```
if key in d:  
    # Yes  
else:  
    # No
```

- Looking up a value that might not exist

```
name = d.get(key, default)
```

- Example:

```
>>> prices.get('IBM', 0.0)  
93.37  
>>> prices.get('SCOX', 0.0)  
0.0  
>>>
```

Composite Keys

- Use tuples

```
holidays = {  
    (1, 1) : 'New Years',  
    (3, 14) : 'Pi day',  
    (9, 13) : "Programmer's day",  
}
```

- Access

```
>>> holidays[3, 14]  
'Pi day'  
>>>
```

Sets

- Sets

```
tech_stocks = { 'IBM', 'AAPL', 'MSFT' }  
tech_stocks = set(['IBM', 'AAPL', 'MSFT'])
```

- A collection of unordered unique items
- Useful for membership tests

```
>>> tech_stocks  
set(['AAPL', 'IBM', 'MSFT'])  
>>> 'IBM' in tech_stocks  
True  
>>> 'FB' in tech_stocks  
False  
>>>
```

Sets

- Sets also useful for duplicate elimination

```
names = ['IBM', 'AAPL', 'GOOG', 'IBM', 'GOOG', 'YHOO']

unique = set(names)
# unique = set(['IBM', 'AAPL', 'GOOG', 'YHOO'])
```

- Other set operations

```
names.add('CAT')           # Add an item
names.remove('YHOO')        # Remove an item

s1 | s2                   # Set union
s1 & s2                   # Set intersection
s1 - s2                   # Set difference
```

Exercise 2.2

Time : 30 minutes

Formatted Output

- When working with data, you often want to produce structured output (tables, etc.).

Name	Shares	Price
AA	100	32.20
IBM	50	91.10
CAT	150	83.44
MSFT	200	51.23
GE	95	40.37
MSFT	50	65.10
IBM	100	70.44

String Formatting

- f-strings (Python 3.6+)

```
name = 'IBM'  
shares = 100  
price = 91.1
```

```
>>> f'{name:>10s} {shares:>10d} {price:>10.2f}'  
'           IBM          100        91.10'  
>>>
```

- $\{expr:fmt\}$ is replaced
- Commonly used with print

```
print(f'{name:>10s} {shares:>10d} {price:>10.2f}')
```

- Format codes are similar to C printf()

Format Codes

d	Decimal integer
b	Binary integer
x	Hexadecimal integer
f	Float as [-]m.dddddd
e	Float as [-]m.dddddde+xx
g	Float, but selective use of E notation
s	String
c	Character (from integer)

Modifiers (partial list)

:>10d	Integer right aligned in 10-character field
:<10d	Integer left aligned in 10-character field
:^10d	Integer centered in 10-character field
:0.2f	Float with 2 digit precision

Dictionary Formatting

- `.format_map()` method of strings

```
s = {  
    'name': 'IBM',  
    'shares': 100,  
    'price': 91.1  
}  
  
>>> '{name:>10s} {shares:10d} {price:10.2f}'.format_map(s)  
'           IBM          100      91.10'  
>>>
```

- Same formatting as f-strings, but takes the values from a supplied dictionary

C-Style Formatting

- Formatting operator (%)

```
>>> 'The value is %d' % 3
'The value is 3'
>>> '%5d %-5d %10d' % (3,4,5)
'      3 4           5'
>>> '%0.2f' % (3.1415926,)
'3.14'
```

- Requires a single item or a tuple on right
- Format codes are modeled after C printf()
- Only formatting available on byte-strings

```
>>> b'%s has %n messages' % (b'Dave', 37)
b'Dave has 37 messages'
>>>
```

Exercise 2.3

Time : 20 minutes

Working with Sequences

- Python has three "sequence" datatypes

```
a = 'Hello'                      # String  
b = [1, 4, 5]                      # List  
c = ('GOOG', 100, 490.1)          # Tuple
```

- Sequences are ordered : $s[n]$

a[0]	→	'H'
b[-1]	→	5
c[1]	→	100

- Sequences have a length : $\text{len}(s)$

len(a)	→	5
len(b)	→	3
len(c)	→	3

Working with Sequences

- Sequences can be replicated : $s * n$

```
>>> a = 'Hello'  
>>> a * 3  
'HelloHelloHello'  
>>> b = [1, 2, 3]  
>>> b * 2  
[1, 2, 3, 1, 2, 3]  
>>>
```

- Similar sequences can be concatenated : $s + t$

```
>>> a = (1, 2, 3)  
>>> b = (4, 5)  
>>> a + b  
(1, 2, 3, 4, 5)  
>>>
```

Sequence Slicing

- Slicing operator : $s[start:end]$

`a = [0,1,2,3,4,5,6,7,8]`

`a[2:5] → [2,3,4]`

0	1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

`a[-5:] → [4,5,6,7,8]`

0	1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

`a[:3] → [0,1,2]`

0	1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

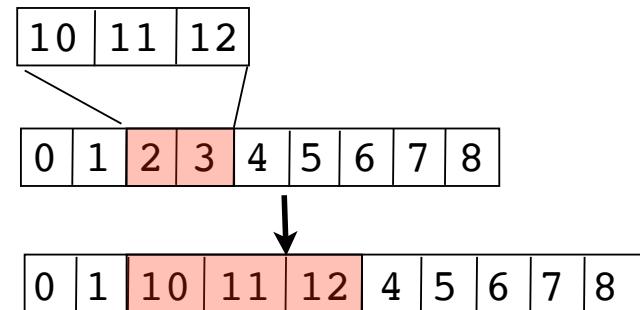
- Indices must be integers
- Slices do not include end value
- If indices are omitted, they default to the beginning or end of the list.

Additional List Slicing

- Slice re-assignment

```
a = [0,1,2,3,4,5,6,7,8]
```

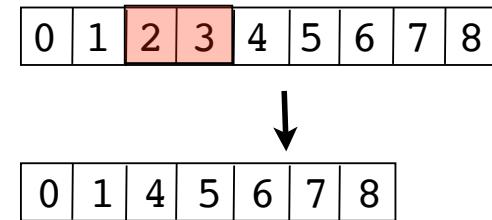
```
a[2:4] = [10,11,12]
```



- Slice deletion

```
a = [0,1,2,3,4,5,6,7,8]
```

```
del a[2:4]
```



Sequence Reductions

- $\text{sum}(s)$

```
>>> s = [1, 2, 3, 4]
>>> sum(s)
10
>>>
```

- $\text{min}(s), \text{max}(s)$

```
>>> min(s)
1
>>> max(s)
4
>>> max(t)
'World'
>>>
```

Iterating over a Sequence

- The for-loop iterates over sequence data

```
>>> s = [1, 4, 9, 16]
>>> for i in s:
...     print(i)
...
1
4
9
16
>>>
```

- On each iteration of the loop, you get new item of data to work with.

Iteration Variables

- Each time through the loop, a new value is placed into an iteration variable

A diagram illustrating a for loop. The code shown is:

```
for x in s:  
    statements:
```

The variable `x` is circled with a dotted line, and an arrow points from this circle to the text "iteration variable" located to the right of the loop body.

iteration variable

- Overwrites the previous value (if any)
- After the loop finishes, the variable retains the last value

break statement

- Breaking out of a loop (exiting)

```
for name in namelist:  
    if name == username:  
        break  
    ...  
    ...  
statements ←
```

- Only applies to the inner-most loop

continue statement

- Skipping to the next iteration

```
for line in lines:  
    if line == '\n':  
        continue  
    # Skip blank lines  
    # More statements  
    ...
```

- Useful if the current item isn't of interest or needs to be ignored in processing

Looping over integers

- If you need to count, use `range()`
- `range([start,] end [,step])`

```
for i in range(100):
    # i = 0,1,...,99

for j in range(10,20):
    # j = 10,11,..., 19

for k in range(10,50,2):
    # k = 10,12,...,48
```

- Note: The ending value is never included
(this mirrors the behavior of slices)

enumerate() Function

- `enumerate(sequence [, start = 0])`
- Provides a loop counter value

```
names = ['Elwood', 'Jake', 'Curtis']
for i, name in enumerate(names):
    # Loops with i = 0, name = 'Elwood'
    #
    #           i = 1, name = 'Jake'
    #
    #           i = 2, name = 'Curtis'
    ...
```

- Example: Keeping a line number

```
with open(filename) as f:
    for lineno, line in enumerate(f, start=1):
        ...
```

enumerate() Function

- `enumerate()` is a nice shortcut

```
for i, x in enumerate(s):  
    statements
```

- Compare to:

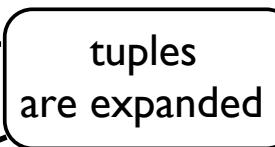
```
i = 0  
for x in s:  
    statements  
    i += 1
```

- Less typing and `enumerate()` runs slightly faster

for and tuples

- Looping with multiple iteration variables

```
points = [  
    (1, 4), (10, 40), (23, 14), (5, 6), (7, 8)  
]  
  
for x, y in points:  
    # Loops with x = 1, y = 4  
    #           x = 10, y = 40  
    #           x = 23, y = 14  
    #           ...
```



tuples
are expanded

- Here, each tuple is unpacked into a set of iteration variables.

zip() Function

- Makes an iterator that combines sequences

```
columns = ['name', 'shares', 'price']
values   = ['GOOG', 100, 490.1]
```

```
pairs = zip(a,b)
# ('name','GOOG'), ('shares',100), ('price',490.1)
```

- To get the result, you must iterate

```
for name, value in pairs:
    ...
```

- Common use: making dictionaries

```
d = dict(zip(columns, values))
```

Exercise 2.4

Time : 15 minutes

collections module

- Contains a few other useful containers
 - defaultdict
 - Counter
 - deque
 - OrderedDict

Counting Things

- Example: Tabulate total shares of each stock

```
portfolio = [  
    ('GOOG', 100, 490.1),  
    ('IBM', 50, 91.1),  
    ('CAT', 150, 83.44),  
    ('IBM', 100, 45.23),  
    ('GOOG', 75, 572.45),  
    ('AA', 50, 23.15)  
]
```

The diagram illustrates the mapping of stock names to their total shares. It shows the original list of tuples on the left and a resulting dictionary on the right. Arrows point from the circled values in the list to the corresponding entries in the dictionary. The circled values are 50, 100, and 100.

Stock Name	Total Shares
'IBM'	150
'GOOG'	175
'AA'	50

- Solution: Use a Counter

```
from collections import Counter  
total_shares = Counter()  
for name, shares, price in portfolio:  
    total_shares[name] += shares  
  
>>> total_shares['IBM']  
150  
>>>
```

One-Many Mappings

- Problem: Map keys to multiple values

```
portfolio = [  
    ('GOOG', 100, 490.1),           {  
    ('IBM', 50, 91.1),             ...  
    ('CAT', 150, 83.44),           'IBM': [ (50, 91.1),  
    ('IBM', 100, 45.23),           (100, 45.23) ]  
    ('GOOG', 75, 572.45),          ...  
    ('AA', 50, 23.15)             }  
]
```

- Solution: Use a defaultdict

```
from collections import defaultdict  
holdings = defaultdict(list)  
for name, shares, price in portfolio:  
    holdings[name].append((shares, price))
```

```
>>> holdings['IBM']  
[ (50, 91.1), (100, 45.23) ]  
>>>
```

Keeping a History

- Problem: Keep a history of the last N things

```
line1  
line2  
line3  
line4  
line5  
...  
history = [ line3, line4, line5 ]
```

- Solution: Use a deque

```
from collections import deque  
  
history = deque(maxlen=N)  
with open(filename) as f:  
    for line in f:  
        history.append(line)  
    ...
```

Exercise 2.5

Time : 10 minutes

List Comprehensions

- Creates a new list by applying an operation to each element of a sequence.

```
>>> a = [1, 2, 3, 4, 5]
>>> b = [2*x for x in a]
>>> b
[2, 4, 6, 8, 10]
>>>
```

- Another example:

```
>>> names = ['Elwood', 'Jake']
>>> a = [name.lower() for name in names]
>>> a
['elwood', 'jake']
>>>
```

List Comprehensions

- A list comprehension can also filter

```
>>> a = [1, -5, 4, 2, -2, 10]
>>> b = [2*x for x in a if x > 0]
>>> b
[2, 8, 4, 20]
>>>
```

- Another example

```
>>> f = open('stockreport', 'r')
>>> goog = [line for line in f if 'GOOG' in line]
>>>
```

List Comprehensions

- General syntax

[*expression* for *names* in *sequence* if *condition*]

- What it means

```
result = []
for names in sequence:
    if condition:
        result.append(expression)
```

- Can be used anywhere a sequence is expected

```
>>> a = [1, 2, 3, 4]
>>> sum([x*x for x in a])
30
>>>
```

List Comp: Examples

- List comprehensions are hugely useful
- Collecting the values of a specific field

```
stocknames = [s['name'] for s in stocks]
```

- Performing database-like queries

```
a = [s for s in stocks if s['price'] > 100  
      and s['shares'] > 50 ]
```

- Data reductions over sequences

```
cost = sum([s['shares']*s['price'] for s in stocks])
```

Historical Digression

- List comprehensions come from math

```
a = [x*x for x in s if x > 0]      # Python
```

```
a = { x2 | x ∈ s, x > 0 }      # Math
```

- Implemented in several other languages
- But most Python programmers would probably just view this as a "cool shortcut"

Exercise 2.6

Time : 15 Minutes

More details on objects

- So far: a tour of the most common types
- Have skipped some critical details
- Memory management
- Copying
- Type checking

The Issue with Assignment

- Many operations in Python are related to "assigning" or "storing" values

```
a = value          # Assignment to a variable  
s[n] = value       # Assignment to an list  
s.append(value)    # Appending to a list  
d['key'] = value   # Adding to a dictionary
```

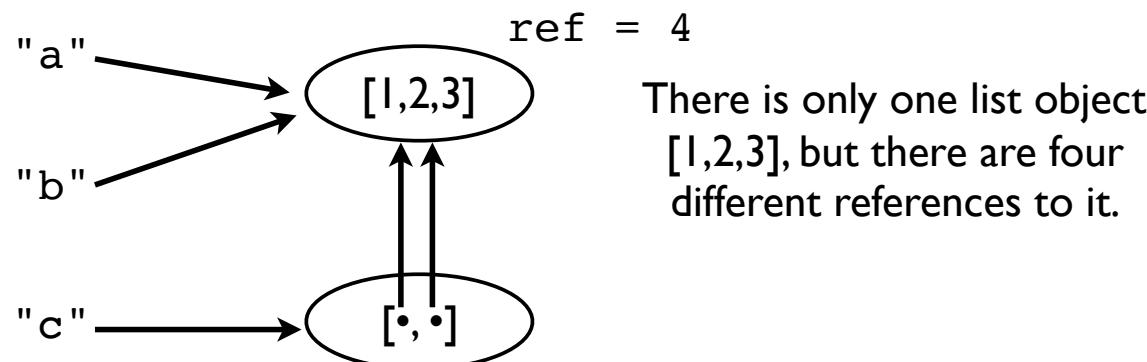
- A caution : assignment operations never make a copy of the value being assigned
- All assignments are merely reference copies (or pointer copies if you prefer)

Assignment Example

- Consider this code fragment:

```
a = [1,2,3]
b = a
c = [a,b]
```

- A picture of the underlying memory



Assignment Example

- Modifying a value affects all references

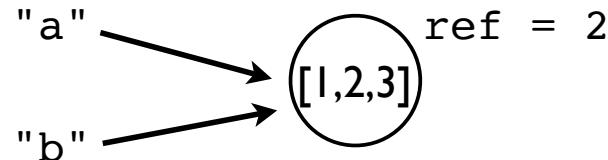
```
>>> a.append(999)
>>> a
[1,2,3,999]
>>> b
[1,2,3,999]
>>> c
[[1,2,3,999], [1,2,3,999]]
>>>
```

- Notice how a change to the original list shows up everywhere else (yikes!)
- This is because no copies were ever made-- everything is pointing at the same thing

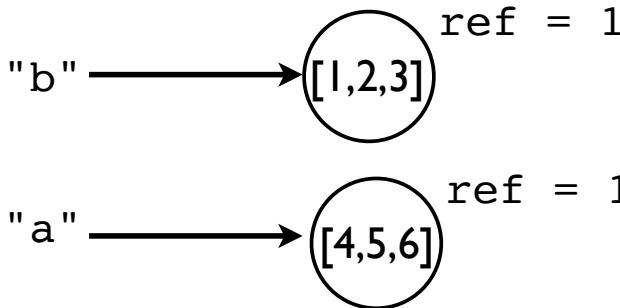
Reassigning Values

- Reassigning a value never overwrites the memory used by the previous value

```
a = [1,2,3]  
b = a
```



```
a = [4,5,6]
```



- Variables are names, not memory locations

Some Dangers

- If you don't know about this sharing, you will shoot yourself in the foot at some point
- Typical scenario : You modify some data thinking that it's your own private copy and it accidentally corrupts some data in some other part of the program
- Comment: This is one of the reasons why the primitive data types (int, float, string) are immutable (read-only)

Identity and References

- Use the "is" operator to check if two values are exactly the same in memory

```
>>> a = [1,2,3]
>>> b = a
>>> a is b
True
>>>
```

- It compares the object identity (an integer)

```
>>> id(a)
3588944
>>> id(b)
3588944
>>>
```

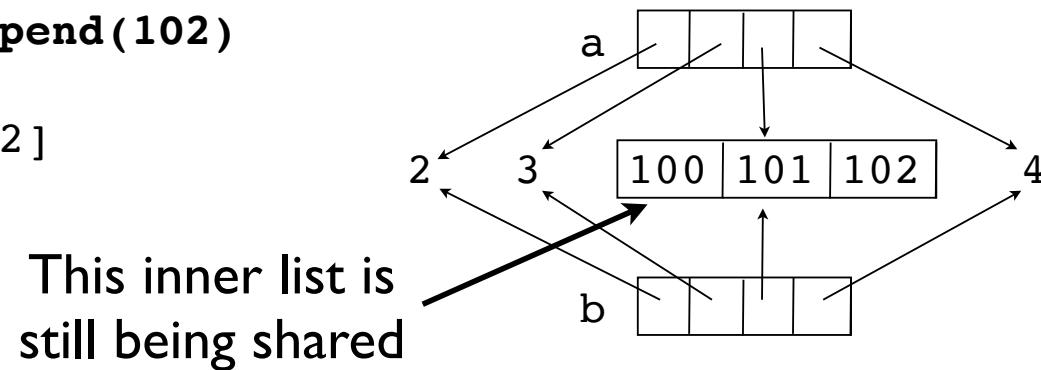
Shallow Copies

- Lists and dicts have methods for copying

```
>>> a = [2,3,[100,101],4]
>>> b = list(a)                      # Make a copy
>>> a is b
False
```

- It's a new list, but the list items are shared

```
>>> a[2].append(102)
>>> b[2]
[100,101,102]
>>>
```



- Known as a "shallow copy"

Deep Copying

- Sometimes you need to makes a copy of an object and all objects contained within it
- Use the `copy` module

```
>>> a = [2,3,[100,101],4]
>>> import copy
>>> b = copy.deepcopy(a)
>>> a[2].append(102)
>>> b[2]
[100,101]
>>>
```

- This is the only safe way to copy something

Names, Values, Types

- Names do not have a "type"--it's only a name
- However, values do have an underlying type

```
>>> a = 42
>>> b = 'Hello World'
>>> type(a)
<type 'int'>
>>> type(b)
<type 'str'>
```

- `type()` will tell you what it is
- The type name is usually a function that creates or converts a value to that type

Type Checking

- How to tell if an object is a specific type

```
if isinstance(a,list):  
    print('a is a list')
```

- Checking for one of many types

```
if isinstance(a,(list,tuple)):  
    print('a is a list or tuple')
```

- Caution: Don't go overboard with type checking (can lead to excessive complexity)

Everything is an object

- Numbers, strings, lists, functions, exceptions, classes, instances, etc...
- All objects are said to be "first-class"
- Meaning: All objects that can be named can be passed around as data, placed in containers, etc., without any restrictions.
- There are no "special" kinds of objects

First Class Objects

- A simple example:

```
>>> import math
>>> items = [abs, math, ValueError] ←
>>> items
[<built-in function abs>,
 <module 'math' (builtin)>,
 <type 'exceptions.ValueError'>]
>>> items[0](-45) ←
45
>>> items[1].sqrt(2)
1.4142135623730951
>>> try:
        x = int('not a number')
    except items[2]: ←
        print('Failed!')
```

A list containing a function, a module, and an exception.

You can use items in the list in place of the original names

Failed!

>>>

Summary

- Have looked at basic principles of working with data in Python programs
- Brief look at part of the object-model
- A big part of understanding most Python programs.

Exercise 2.7

Time : 15 Minutes

Section 3

Program Organization

Overview

- How to organize larger programs
- More details on program execution
- Defining and working with functions
- Exceptions and Error Handling

Observation

- A large number of Python programmers spend most of their time writing short "scripts"
- One-off problems, prototyping, testing, etc.
- Python is good at this!
- And it what draws many users to Python

What is a "Script?"

- A "script" is a program that runs a series of statements and stops

```
# program.py
```

```
statement1  
statement2  
statement3  
...
```

- We've been writing scripts to this point

Problem

- If you write a useful script, it will grow features
- You may apply it to other related problems
- Over time, it might become a critical application
- And it might turn into a huge tangled mess
- So, let's get organized...

Defining Things

- You must always define things before they get used later on in a program.

```
def square(x):  
    return x*x  
  
a = 42  
b = a + 2      # Requires that a is already defined  
  
z = square(b)  # Requires square to be defined
```

- The order is important
- You almost always put the definitions of variables and functions near the beginning

Defining Functions

- It is a good idea to put all of the code related to a single "task" all in one place

```
def read_prices(filename):  
    prices = {}  
    with open(filename) as f:  
        f_csv = csv.reader(f)  
        for row in f_csv:  
            prices[row[0]] = float(row[1])  
    return prices
```

- A function also simplifies repeated operations

```
oldprices = read_prices('oldprices.csv')  
newprices = read_prices('newprices.csv')
```

What is a function?

- A function is a sequence of statements

```
def funcname(args):  
    statement  
    statement  
    ...  
    return result
```

- Any Python statement can be used inside

```
def foo():  
    import math  
    print(math.sqrt(2))  
    help(math)
```

- There are no "special" statements in Python

Function Definitions

- Functions can be defined in any order

```
def foo(x):  
    bar(x)  
  
def bar(x):  
    statements  
  
def bar(x):  
    statements  
  
def foo(x):  
    bar(x)
```

- Functions must only be defined before they are actually used during program execution

```
foo(3)      # foo must be defined already
```

- Stylistically, it is more common to see functions defined in a "bottom-up" fashion

Bottom-up Style

- Functions are treated as building blocks
- The smaller/simpler blocks go first

```
# myprogram.py
def foo(x):
    ...
def bar(x):  
    ...
    foo(x)
    ...
def spam(x):
    ...
    bar(x)
    ...
spam(42)      # Call spam() to do something
```

Later functions build upon earlier functions

Code that uses the functions appears at the end

Function Design

- Try to make functions that only operate on their inputs and which return a proper result

Yes

```
def read_prices(filename):
    prices = {}
    with open(filename) as f:
        ...
    return prices

prices = read_prices('prices.csv')
```

No

```
filename = 'prices.csv'
prices = {}

def read_prices()
    with open(filename) as f:
        ...
    return
```

- Depending on external variables makes everything worse. Don't do that.

Doc Strings

- Good practice to include documentation

```
def read_prices(filename):
    """
        Read prices from a CSV file of name,price
    """

    prices = {}
    with open(filename) as f:
        f_csv = csv.reader(f)
        for row in f_csv:
            prices[row[0]] = float(row[1])
    return prices
```

- Doc-string feeds help(), IDEs, and other tools

Type Annotations

- Might see optional type annotations

```
def read_prices(filename: str) -> dict:  
    ...  
    Read prices from a CSV file of name,price  
    ...  
    prices = {}  
    with open(filename) as f:  
        f_csv = csv.reader(f)  
        for row in f_csv:  
            prices[row[0]] = float(row[1])  
    return prices
```

- These do nothing. Purely informational
- May be used by IDEs, code checkers, etc.

Exercise 3. I

Time : 15 minutes

Default Arguments

- Sometimes you want an optional argument

```
def read_prices(filename, debug=False):  
    ...
```

- If a default value is assigned, the argument is optional in function calls

```
d = read_prices('prices.csv')  
e = read_prices('prices.dat', True)
```

- Note :Arguments with defaults must appear at the end of the argument list (all non-optional arguments go first)

Calling a Function

- Consider a simple function

```
def read_prices(filename, debug):  
    ...
```

- Calling with "positional" args

```
prices = read_prices('prices.csv', True)
```

- Calling with "keyword" arguments

```
prices = read_prices(filename='prices.csv',  
                     debug=True)
```

Keyword Arguments

- Keyword arguments are useful for functions that have optional features/flags

```
def parse_data(data, debug=False, ignore_errors=False):  
    ...
```

- Compare and contrast

```
parse_data(data, False, True)      # ????
```

```
parse_data(data, ignore_errors=True)  
parse_data(data, debug=True)  
parse_data(data, debug=True, ignore_errors=True)
```

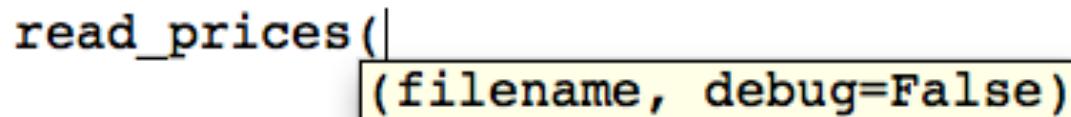
- Keyword arguments improve code clarity

Design Tip

- Always give short, but meaningful names to function arguments
- Someone using a function may want to use the keyword calling style

```
d = read_prices('prices.csv', debug=True)
```

- Python development tools will show the names in help features and documentation



```
read_prices(  
    filename, debug=False)
```

Return Values

- return statement returns a value

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

- If no return value, `None` is returned

```
def bar(x):  
    statements  
    return  
  
a = bar(4)      # a = None
```

Multiple Return Values

- A function may return multiple values by returning a tuple

```
def divide(a,b):  
    q = a // b      # Quotient  
    r = a % b      # Remainder  
    return q, r    # Return a tuple
```

- Usage example:

```
x, y = divide(37, 5)      # x = 7, y = 2  
  
x = divide(37, 5)          # x = (7, 2)
```

Understanding Variables

- Programs assign values to variables

```
x = value          # Global variable
```

```
def foo():  
    y = value      # Local variable
```

- Variable assignments occur outside and inside function definitions
- Variables defined outside are "global"
- Variables inside a function are "local"

Local Variables

- Variables inside functions are private

```
def read_portfolio(filename):  
    portfolio = []  
    for line in open(filename):  
        fields = line.split()  
        s = (fields[0],int(fields[1]),float(fields[2]))  
        portfolio.append(s)  
    return portfolio
```

- Values not retained or accessible after return

```
>>> stocks = read_portfolio('stocks.dat')  
>>> fields  
Traceback (most recent call last):  
  File "<stdin>", line 1, in ?  
NameError: name 'fields' is not defined  
>>>
```

- Don't conflict with variables found elsewhere

Global Variables

- Functions can access the values of globals

```
name = 'Dave'  
def greeting():  
    print('Hello', name)
```

- A quirk: Functions can't modify globals

```
def spam():  
    name = 'Guido'  
  
spam()  
print(name)      # prints 'Dave'
```

- All assignments in functions are local

Modifying Globals

- If you must modify a global variable you must declare it as such

```
name = 'Dave'

def spam():
    global name
    name = 'Guido' # Changes the global name above
```

- **global declaration must appear before use**
- Considered "bad style"
- Avoid entirely if you can (use a class instead)

Argument Passing

- When you call a function, the argument variables are names for passed values
- If mutable data types are passed (e.g., lists, dicts), they can be modified "in-place"

```
def foo(items):  
    items.append(42) ← Modifies the  
    a = [1, 2, 3] ← input object  
    foo(a)  
    print(a)      # [1, 2, 3, 42]
```

- Key point: Function doesn't receive a copy

Understanding Assignment

- Make sure you understand the subtle difference between modifying a value and reassigning a variable name
- Example:

```
def foo(items):  
    items.append(42) # Modifies items list  
  
def bar(items):  
    items = [4,5,6] # Binds name 'items' to new list
```

- Reminder : Variable assignment never overwrites memory (the name is simply bound to a new value)

Exercise 3.2

Time : 30 minutes

Error Checking

- Python performs no checking or validation of function argument types or values
- A function will work on any data that is compatible with the statements in the function
- Example:

```
def add(x, y):  
    return x + y  
  
add(3, 4)          # 7  
add('Hello', 'World')  # 'HelloWorld'  
add('3', '4')       # '34'
```

Error Checking

- If there are errors in a function, they will show up at run time (as an exception)
- Example:

```
def add(x, y):  
    return x + y  
  
>>> add(3, '4')  
Traceback (most recent call last):  
...  
TypeError: unsupported operand type(s) for +:  
'int' and 'str'  
>>>
```

- To verify code, there is a strong emphasis on testing (covered later)

Exceptions

- Used to signal errors
- Raising an exception (`raise`)

```
if name not in names:  
    raise RuntimeError('Name not found')
```

- Catching an exception (`try`)

```
try:  
    authenticate(username)  
except RuntimeError as e:  
    print(e)
```

Exceptions

- Exceptions propagate to first matching except

```
def foo():
    try:
        bar()
    except RuntimeError as e:
        ...

def bar():
    try:
        spam()
    except RuntimeError as e:
        ...
    def spam():
        grok()

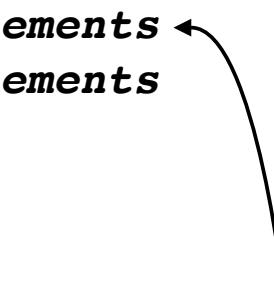
def grok():
    ...
    raise RuntimeError('Whoa!')
```

The diagram illustrates the propagation of an exception. It shows four code snippets: foo(), bar(), spam(), and grok(). An arrow points from the 'raise' statement in grok() up to the 'except' clause in bar(). Another arrow points from the 'except' clause in bar() up to the 'except' clause in foo(). This visualizes how an exception raised in a local scope (grok()) propagates through multiple levels of scopes until it finds a matching 'except' clause.

Exceptions

- To handle the exception, statements inside the `except` block run

```
def bar():
    try:
        grok()
    except RuntimeError as e:
        statements ←
        statements
        ...
def grok():
    ...
    raise RuntimeError('Whoa!')
```



Exceptions

- After handling, execution resumes with the first statement after the try-except

```
def bar():
    try:
        grok()
    except RuntimeError as e:
        statements
        statements
        ...
        statements
        statements
        ...
def grok():
    ...
    raise RuntimeError("Whoa!")
```

The diagram illustrates the flow of execution. It starts with a code snippet for the `bar()` function. Inside the `try` block, there is a call to `grok()`. Following this, there is a `except` block that handles a `RuntimeError`. Within this exception block, there are two lines of code labeled **statements**, followed by three ellipses, and then two more lines of code labeled **statements**, followed by another three ellipses. A vertical arrow points downwards from the end of the `except` block to the start of the `grok()` function definition below it.

Builtin-Exceptions

- About two-dozen built-in exceptions

- ArithmetricError
- AssertionError
- EnvironmentError
- EOFError
- ImportError
- IndexError
- KeyboardInterrupt
- KeyError
- MemoryError
- NameError
- ReferenceError
- RuntimeError
- SyntaxError
- SystemError
- TypeError
- ValueError

- Consult reference

Exception Values

- Most exceptions have an associated value
- More information about what's wrong

```
raise RuntimeError('Invalid user name')
```

- Passed to variable supplied in except

```
try:  
    ...  
except RuntimeError as e:  
    ...
```

- It's an instance of the exception type, but often looks like a string

```
except RuntimeError as e:  
    print('Failed : Reason', e)
```

Catching Multiple Errors

- Can catch different kinds of exceptions

```
try:  
    ...  
except LookupError as e:  
    ...  
except RuntimeError as e:  
    ...  
except IOError as e:  
    ...  
except KeyboardInterrupt as e:  
    ...
```

- Alternatively, if handling is same

```
try:  
    ...  
except (IOError,LookupError,RuntimeError) as e:  
    ...
```

Catching All Errors

- Catching any exception

```
try:  
    ...  
except Exception:  
    print('An error occurred')
```

- A really bad idea as shown (don't do it!)

Exploding Heads

- The wrong way to use exceptions:

```
try:  
    go_do_something()  
except Exception:  
    print('Computer says no')
```

- This swallows all possible errors
- May make it impossible to debug if code is failing for some reason you didn't expect at all (e.g., uninstalled Python module, etc.)

A Better Approach

- This is a somewhat more sane approach

try:

go_do_something()

except Exception as e:

print('Computer says no. Reason :', e)

- Reports a specific reason for the failure
- It is almost always a good idea to have some mechanism for viewing/reporting errors if you are writing code that catches all possible exceptions

Reraising an Exception

- Use 'raise' to propagate a caught error

```
try:  
    go_do_something()  
except Exception as e:  
    print('Computer says no. Reason :', e)  
    raise
```

- Allows you to take action (e.g., logging), but pass the error on to the caller

Exception Advice

- Don't catch exceptions - fail fast and loud
(if it's important, someone else will take care of the problem)
- Only catch an exception if you're *that* someone
- That is, only catch errors where you can recover and sanely keep going

finally statement

- Specifies code that must run regardless of whether or not an exception occurs

```
lock = Lock()  
...  
lock.acquire()  
try:  
    ...  
finally:  
    lock.release()      # release the lock
```

- Commonly use to properly manage resources (especially locks, files, etc.)

with statement

- In modern code, try-finally often replaced with the 'with' statement

```
lock = Lock()
with lock:
    # lock acquired
    ...
# lock released

with open(filename) as f:
    # Use the file
    ...
# File closed
```

- Defines a usage "context" for a resource
- Only works with certain objects

Program Exit

- Program exit is handle through exceptions

```
raise SystemExit  
raise SystemExit(exitcode)  
raise SystemExit('Informative message')
```

- An alternative sometimes seen

```
import sys  
sys.exit()  
sys.exit(exitcode)
```

- Hard exit (immediate, no cleanup)

```
import os  
os._exit(exitcode)
```

Exercise 3.3

Time : 15 minutes

Modules

- Any Python source file is a module

```
# foo.py
def grok(a):
    ...
def spam(b):
    ...
```

- import statement loads and executes a module

```
import foo

a = foo.grok(2)
b = foo.spam('Hello')
...  
...
```

Namespaces

- A module is a collection of named values (i.e., it's said to be a "namespace")
- The names are all of the global variables and functions defined in the source file
- After import, module name used as a prefix

```
>>> import foo  
>>> foo.grok(2)  
>>>
```

- Module name is tied to source (foo -> foo.py)

Global Definitions

- Everything defined in the "global" scope is what populates the module namespace

```
# foo.py  
x = 42  
def grok(a):  
    ...
```

```
# bar.py  
x = 37  
def spam(a):  
    ...
```

These definitions of x
are different

- Different modules can use the same names and those names don't conflict with each other (modules are isolated)

Modules as Environments

- Modules form an enclosing environment for all of the code defined inside

```
# foo.py
x = 42
def grok(a):
    print(x)
```



global variables are always
bound to the enclosing
module (same file)

- Each source file is its own little universe
- This is great!
- What happens in a module stays in a module

Module Execution

- When a module is imported, all of the statements in the module execute one after another until the end of the file is reached
- The contents of the module namespace are all of the global names that are still defined at the end of the execution process
- If there are scripting statements that carry out tasks in the global scope (printing, creating files, etc.), you will see them run on import

import as statement

- Changing the name of a module

```
import math as m

def rectangular(r, theta):
    x = r * m.cos(theta)
    y = r * m.sin(theta)
    return x, y
```

- Same as a normal import
- It renames the module in that one file (the one that did the import)

from module import

- Lifts selected symbols out of a module and makes them available locally

```
from math import sin, cos
```

```
def rectangular(r, theta):  
    x = r * cos(theta)  
    y = r * sin(theta)  
    return x, y
```

- Allows parts of a module to be used without having to type the module prefix
- Useful for frequently used features

Commentary

- Variations on import do not change the way that modules work

```
import math as m  
from math import cos, sin  
...
```

- import always executes the entire file
- Modules are still isolated environments
- These variations are only manipulating names

Module Loading

- Each module loads and executes once
- Repeated imports just return a reference to the previously loaded module
- `sys.modules` is a dict of all loaded modules

```
>>> import sys
>>> sys.modules.keys()
['copy_reg', '__main__', 'site', '__builtin__',
'encodings', 'encodings.encodings', 'posixpath', ...]
>>>
```

Locating Modules

- When looking for modules, Python consults a path list (`sys.path`)

```
>>> import sys  
>>> sys.path  
[  
    '',  
    '/usr/local/lib/python36/python36.zip',  
    '/usr/local/lib/python36',  
    ...  
]
```

- Current working directory is usually first

Module Search Path

- `sys.path` contains search path
- Can manually adjust if you need to

```
import sys
sys.path.append('/project/foo/pyfiles')
```

- Paths also added via environment variables

```
% env PYTHONPATH=/project/foo/pyfiles python3
Python 3.6.0 (default, Feb  3 2017, 05:53:21)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.38)]
>>> import sys
>>> sys.path
[ '', '/project/foo/pyfiles', ... ]
```

Exercise 3.4

Time : 15 Minutes

Main Functions

- In many programming languages, there is a concept of a "main" function or method

```
/* C/C++ */  
int main(int argc, char *argv[]) {  
    ...  
}  
  
/* Java */  
class myprog {  
    public static void main(String args[]) {  
        ...  
    }  
}
```

- It's the first function that executes when an application is launched

Main Module

- Python has no "main" function or method
- Instead, there is a "main" module
- It's the source file that runs first

```
bash % python3 foo.py  
...
```

- Whatever module you give to the interpreter at startup becomes "main"

__main__ check

- It is standard practice for modules that can run as a main script to use this convention:

```
# foo.py
...
if __name__ == '__main__':
    # Running as the main program
    ...
    statements
    ...
```

- Statements enclosed inside the if-statement become the "main" program

__main__ check

- Important: Any file can either run as main or as a library import

```
bash % python3 foo.py      # Running as main  
  
>>> import foo           # Loaded as a module
```

- __name__ is the name of the module
- As a general rule, you don't want statements that are part of a main program to execute on a library import (hence, the check)

```
if __name__ == '__main__':  
    # Does not execute if loaded with import  
    ...
```

A Script Template

```
#!/usr/bin/env python
# script.py

# Import statements (libraries)
import modules

# Functions
def spam():
    ...

def blah():
    ...

# Main function
def main():
    ...

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

sys: Command Line Opt

- Programs usually execute from the shell

```
bash % python3 report.py portfolio.csv prices.csv
```

- Arguments get placed in sys.argv

```
sys.argv → ['report.py', 'portfolio.csv', 'prices.csv']
```

- Example of processing

```
import sys
if len(sys.argv) != 3:
    raise SystemExit(f'Usage: {sys.argv[0]} '
                    'portfile pricefile')
portfile = sys.argv[1]
pricefile = sys.argv[2]
...
```

Advanced Arguments

- Use argparse for advanced argument parsing

```
import argparse
import sys

parser = argparse.ArgumentParser()
parser.add_argument('-o', '--output')
parser.add_argument('-d', '--debug', action='store_true')
parser.add_argument('portfile')
parser.add_argument('pricefile')

# Parse the options
args = parser.parse_args(sys.argv)

# Retrieve the option values
portfile = args.portfile
pricefile = args.pricefile
outfile = args.output
debug = args.debug
```

Revised Script Template

```
#!/usr/bin/env python
# script.py

# Import statements (libraries)
import modules

# Functions
def spam():
    ...

def blah():
    ...

# Main function
def main(args):
    # Parse command line args (argparse, etc.)
    ...

if __name__ == '__main__':
    import sys
    main(sys.argv)
```

Exercise 3.5

Time : 10 Minutes

Standard Library

- Python includes a large standard library
- Several hundred modules
- System, networking, data formats, etc.
- All accessible via import
- Let's take a tour of the most essential ones

sys module

- Information related to environment
- Version information
- System limits
- Command line options
- Module search paths
- Standard I/O streams

sys: Standard I/O

- Standard I/O streams

```
sys.stdout  
sys.stderr  
sys.stdin
```

- By default, print is directed to sys.stdout
- Input read from sys.stdin
- Can redefine or use directly

```
sys.stdout = open('out.txt', 'w')  
print('Warning. Unable to connect', file=sys.stderr)
```

sys: Standard I/O

- Function that allows I/O redirection

```
import sys
def greet(name, outfile=None):
    if outfile is None:
        outfile = sys.stdout
    outfile.write(f'Hello {name}\n')
```

- Example:

```
# Write to standard out
greet('Dave')

# Write to a file
f = open('somefile.txt', 'w')
greet('Dave', outfile=f)
f.close()
```

os Module

- Contains operating system functions
- Example: Executing a shell command

```
>>> import os  
>>> os.system('mkdir temp')  
>>>
```

- Example: Executing a system call
- Hundreds of other low-level operations

Environment Variables

- Environment variables (typically set in shell)

```
% setenv NAME dave  
% setenv RSH ssh  
% python prog.py
```

- os.environ dictionary contains values

```
import os  
home = os.environ[ 'HOME' ]  
os.environ[ 'HOME' ] = '/home/user/guest'
```

- Changes are reflected in Python and any subprocesses created later

Getting a Directory Listing

- `os.listdir()` function

```
>>> files = os.listdir('/some/path')
>>> files
['foo', 'bar', 'spam']
>>>
```

- `glob module`

```
>>> txtfiles = glob.glob('*.*txt')
>>> datfiles = glob.glob('Dat[0-5]*')
>>>
```

- `glob` understands Unix shell wildcards (on all systems)

os.path Module

- Portable management of path names and files
- Examples:

```
>>> import os.path
>>> os.path.basename('/home/foo/bar.txt')
'bar.txt'
>>> os.path.dirname('/home/foo/bar.txt')
'/home/foo'
>>> os.path.join('home', 'foo', 'bar.txt')
'home/foo/bar.txt'
>>>
```

- Solves problem of '/' vs. '\' on Unix/Windows

File Tests

- Testing if a file exists

```
>>> os.path.exists('foo.txt')
True
>>>
```

- Testing if a filename is a regular file

```
>>> os.path.isfile('foo.txt')
True
>>> os.path.isfile('/usr')
False
>>>
```

- Testing if a filename is a directory

```
>>> os.path.isdir('foo.txt')
False
>>> os.path.isdir('/usr')
True
>>>
```

Pathnames

- Expand into a full absolute path name

```
>>> os.path.abspath('README.html')
'/Users/beazley/Desktop/Training/Exercises/
PythonClass/README.html'
>>>
```

- Relative paths (to current working directory)

```
>>> os.path.relpath('/etc/passwd')
'../../../../../etc/passwd'
>>>
```

- Expanding user home directory (~)

```
>>> os.path.expanduser '~/Desktop'
'/Users/beazley/Desktop'
>>>
```

File Metadata

- Getting the file size

```
>>> os.path.getsize('foo.txt')
1344L
>>>
```

- Getting the last modification/access time

```
>>> os.path.getmtime('foo.txt')
1175769416.0
>>> os.path.getatime('foo.txt')
1175769491.0
>>>
```

- Note: To decode times, use time module

```
>>> time.ctime(os.path.getmtime('foo.txt'))
'Thu Apr  5 05:36:56 2007'
>>>
```

Directory Walking

- Walking over a directory tree

```
for path, dirs, files in os.walk(topdir):
    # path = name of current directory
    # dirs = list of all subdirectories in path
    # files = list of all files in path

    # Example: Print out names of .py files
    for filename in files:
        if filename.endswith('.py'):
            print(os.path.join(path,filename))
```

- `walk()` is a little tricky to use at first, but it can be used to carry out operations similar to those performed with Unix 'find'

Shell Operations (shutil)

- Copying a file

```
>>> shutil.copy('source', 'dest')
```

- Moving a file (renaming)

```
>>> shutil.move('old', 'new')
```

- Copying a directory tree

```
>>> shutil.copytree('srcdir', 'destdir')
```

- Removing a directory tree

```
>>> shutil.rmtree('dirname')
```

time module

- System date and time related functions

```
time.sleep(seconds)
time.clock()                      # CPU time (seconds)
time.time()                        # Real time (seconds)
time.localtime([secs])          # Time as a struct
time.ctime([tmstruct])          # Time as a string
```

- Example

```
>>> time.time()
1267128590.589685
>>> time.localtime()
time.struct_time(tm_year=2010, tm_mon=2,
tm_mday=25, tm_hour=14, tm_min=9, tm_sec=54,
tm_wday=3, tm_yday=56, tm_isdst=0)
>>> time.ctime()
'Thu Feb 25 14:09:58 2010'
>>>
```

datetime module

- A module for more generic representation and manipulation of dates and times

```
>>> from datetime import datetime
>>> cataclysm = datetime(2012,12,21)
>>> cataclysm
datetime.datetime(2012, 12, 21, 0, 0)
>>> today = datetime.today()
>>> d = cataclysm - today
>>> d
datetime.timedelta(1249, 32536, 964510)
>>> d.days
1249
>>>
```

- There are many more features not shown

subprocess Module

- A module for launching subprocesses
- Capture output of a command as string

```
import subprocess
out = subprocess.check_output(['ls', '-l'])
```

- Low-level subprocess with stdin/stdout pipes

```
p = subprocess.Popen(['wc'],
                     stdout=subprocess.PIPE,
                     stdin=subprocess.PIPE)
p.stdin.write('Hello World\n')
p.stdin.close()
data = p.stdout.read()
```

re Module

- Support for regular expression patterns

```
>>> import re
>>> text = 'Today is 1/17/2014. Tomorrow is 1/18/2014'

>>> # Find all matches of a pattern
>>> re.findall(r'(\d+)/(\d+)/(\d+)', text)
[('1', '17', '2014'), ('1', '18', '2014')]

>>> # Replace a pattern with new text
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Today is 2014-1-17. Tomorrow is 2014-1-18'

>>> # Splitting on a pattern
>>> re.split(r'[,:]', '1,2:3,4')
['1', '2', '3', '4']
>>>
```

- Several other operations, but this is basic idea

Regex Pattern Syntax

- Regular expression pattern syntax overview

```
foo          # Matches the text "foo"
(foo|bar)    # Matches the text "foo" or "bar"
(foo)*       # Match 0 or more repetitions of foo
(foo)+       # Match 1 or more repetitions of foo
(foo)?       # Match 0 or 1 repetitions of foo
(foo){N}      # Match N repetitions of foo
[abcde]      # Match one of the letters a,b,c,d,e
[a-z]        # Match one letter from a,b,...,z
[^a-z]        # Match any character except a,b,...z
.             # Match any character except newline
$             # Match end of line
\*            # Match the * character
\+            # Match the + character
\d            # Match a digit
\s            # Match whitespace
\w            # Match alphanumeric character
```

- Many other advanced options (not shown)

re: Groups

- Parenthesized parts of a pattern define groups

```
pat = r'(\d+)/(\d+)/(\d+)'
```

- Groups are assigned numbers

```
pat = r'(\d+)/(\d+)/(\d+)'  
      ↑   ↑   ↑  
      1   2   3
```

- Numbering is determined by looking at '(' from left to right

re: Groups

- When searching, groups get separated

```
>>> re.findall(r'(\d+)/(\d+)/(\d+)', text)
[('1', '17', '2014'), ('1', '18', '2014')]
>>>
          ↑   ↑   ↑
      group 1  group 2  group 3
```

- Use \N to refer to groups in replacements

```
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Today is 2014-1-17. Tomorrow is 2014-1-18'
>>>
```

Writing Regex Patterns

- Patterns are written out as strings
- Usually using raw strings because the '\' character has meaning in regex patterns
- Example

```
pat = r'(\d+)/(\d+)/(\d+)'
```

- Recall raw strings don't interpret escapes (\)

JSON Encoding/Decoding

- JSON - JavaScript Object Notation

```
{  
  "recipe" : {  
    "title" : "Famous Guacamole",  
    "description" : "A southwest favorite!",  
    "ingredients" : [  
      {"num": "2", "item":"Large avocados, chopped"},  
      {"num": "1/2", "units":"C", "item":"White onion, chopped"},  
      {"num": "1", "units":"tbl", "item":"Fresh squeezed lemon juice"},  
      {"num": "1", "item":"Jalapeno pepper, diced"},  
      {"num": "1", "units":"tbl", "item":"Fresh cilantro, minced"},  
      {"num": "3", "units":"tsp", "item":"Sea Salt"},  
      {"num": "6", "units":"bottles", "item":"Ice-cold beer"}  
    ],  
    "directions" : "Combine all ingredients and hand whisk to desired  
consistency. Serve and enjoy with ice-cold beers."  
  }  
}
```

- Extremely common in web-services

JSON-Dict Translation

- Converting a dict to JSON

```
>>> import json
>>> s = {'name': 'GOOG', 'shares': 100, 'price': 490.1}
>>> encoded = json.dumps(s)
>>> encoded
'{"price": 490.1, "name": "GOOG", "shares": 100}'
>>>
```

- Converting JSON into a dict

```
>>> json.loads(encoded)
{'price': 490.1, 'name': 'GOOG', 'shares': 100}
>>>
```

Exercise 3.6

Time : 15 Minutes

Section 4

Classes and Objects

OO in a Nutshell

- A programming technique where code is organized as a collection of "objects"
- An "object" consists of
 - Data (attributes)
 - Methods (functions applied to object)
- You've already been doing it

OO in a Nutshell

- Example: Lists

```
>>> nums = [1, 2, 3]
>>> nums.append(4)          # Method
>>> nums.insert(1,10)       # Method
>>>
```

- `nums` is an "instance" of a list
- methods are attached to the instance

The class statement

- How to define your own custom objects

```
class Circle(object):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * (self.radius ** 2)

    def perimeter(self):
        return 2 * math.pi * self.radius
```

- What is a class?
- It's a collection of functions that perform various operations on instances

Instances

- Created by calling the class as a function

```
>>> c = Circle(4.0)
>>> d = Circle(5.0)
>>>
```

- Each instance has its own data

```
>>> c.radius
4.0
>>> d.radius
5.0
>>>
```

- You invoke methods on instances to do things

```
>>> c.area()
50.26548245743669
>>> d.perimeter()
31.415926535897931
>>>
```

__init__ method

- This method initializes a new instance
- Called whenever a new object is created

```
>>> c = Circle(4.0)  
  
class Circle(object):  
    def __init__(self, radius):  
        self.radius = radius  
  
newly created object
```

- __init__ is example of a "special method"
- Has special meaning to Python interpreter

Instance Data

- Each instance has its own data (attributes)

```
class Circle(object):
    def __init__(self, radius):
        self.radius = radius
```

- Inside methods, you refer to this data using `self`

```
def area(self):
    return math.pi * (self.radius ** 2)
```

- In other code, you just use the variable that you're using to name the instance

```
>>> c = Circle(4.0)
>>> c.radius
4.0
```

Methods

- Functions applied to instances of an object

```
class Circle(object):  
    ...  
    def area(self):  
        return math.pi * (self.radius ** 2)
```

- The object is always passed as first argument

```
>>> c.area()  
def area(self):  
    ...
```

- By convention, the instance is called "self"

The name is unimportant---the object is always passed as the first argument. It is simply Python programming style to call this argument "self." It's similar to "this" in C++/Java.

Calling Other Methods

- Methods call other methods via self

```
class Circle(object):
    def area(self):  
        return math.pi * (self.radius ** 2)  
  
    def print_area(self):  
        print(self.area())
```

- A caution : Code like this doesn't work

```
class Circle(object):
    ...  
    def print_area(self):  
        print(area())          # ! Error
```

- It calls a global function area() instead

Exercise 4. I

Time : 15 Minutes

Inheritance

- A tool for specializing objects

```
class Parent(object):  
    ...  
  
class Child(Parent):  
    ...
```

- New class called a derived class or subclass
- Parent known as base class or superclass
- Parent is specified in () after class name

Inheritance

- What do you mean by "specialize?"
- Take an existing class and ...
 - Add new methods
 - Redefine some of the existing methods
 - Add new attributes to instances

Inheritance Example

- In bill #246 of the 1897 Indiana General Assembly, there was text that dictated a new method for squaring a circle, which if adopted, would have equated π to 3.2.
- Fortunately, it was never adopted because an observant mathematician took notice...
- But, let's make a special Indiana Circle anyways...

Inheritance Example

- Specializing a class

```
class INCircle(Circle):  
    def area(self):  
        return 3.2 * (self.radius ** 2)
```

- Using the specialized version

```
>>> c = INCircle(4.0)      # Calls Circle.__init__  
>>> c.radius  
4.0  
>>> c.area()              # Calls INCircle.area  
51.20  
>>> c.perimeter()         # Calls Circle.perimeter  
25.132741228718345  
>>>
```

- It's the same as Circle except for area()

Calling Other Methods

- With inheritance, the correct method gets called if overridden (depends on the type of self)

```
class Circle(object):
    def area(self):
        return math.pi * (self.radius ** 2)

    def print_area(self):
        print(self.area())

class INCircle(Circle):
    def area(self): ←----- if self is an
        return 3.2 * (self.radius ** 2) instance of
                                            INCircle
```

- Example:

```
>>> c = INCircle(4)
>>> c.print_area()
51.2
>>>
```

Using Inheritance

- Sometimes used to organize related objects

```
class Shape(object):  
    ...  
  
class Circle(Shape):  
    ...  
  
class Rectangle(Shape):  
    ...
```

- Think logical hierarchy or taxonomy

Using Inheritance

- More commonly used as a code reuse tool

```
class CustomHandler(TCPServer):  
    def handle_request(self):  
        ...  
        # Custom processing
```

- Base class contains general purpose code
- You inherit to customize specific parts
- Maybe it plugs into a framework

"is a" relationship

- Inheritance establishes a type relationship

```
class Shape(object):
    ...
class Circle(Shape):
    ...
>>> c = Circle(4.0)
>>> isinstance(c, Shape)
True
>>>
```

- Important: Code that works with the parent is also supposed to work with the child

object base class

- If a class has no parent, use object as base

```
class Shape(object):  
    ...
```

- object is the parent of all objects in Python
- Note : It's not technically required in Python 3, but if omitted in Python 2, it results in an "old style class" which should be avoided.

Inheritance and Overriding

- Sometimes a class extends an existing method, but it wants to use the original implementation

```
class Parent(object):
    def spam(self): ←
        ...
    ...
class Child(Parent):
    def spam(self):
        ...
        r = super().spam() # super(Child, self) in Py2
        ...
```

notice how both methods have the same name.

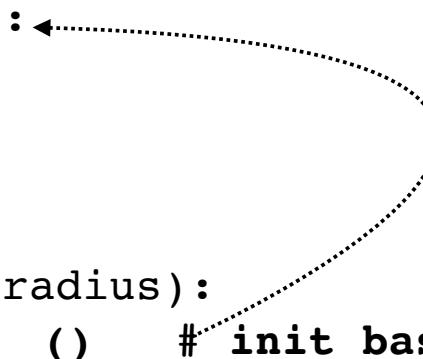
- Use `super()` to do it
- In Python 2: Use `super(cls, self)`

Inheritance and `__init__`

- With inheritance, you must initialize parents

```
class Shape(object):
    def __init__(self):  
        self.x = 0.0  
        self.y = 0.0  

    ...  
class Circle(Shape):
    def __init__(self, radius):  
        super().__init__() # init base  
        self.radius = radius
```



- Again, you should use `super()` as shown

Overriding Caution

- Sometimes you will see overrides implemented with a direct call to the parent class

```
class Parent(object):
    def spam(self): ←
        ...
        ...
class Child(Parent):
    def spam(self):
        ...
        r = Parent.spam(self)
        ...
```



direct call to parent

- This is an older style with subtle limitations
- Usually better to use super()

Multiple Inheritance

- You can specifying multiple base classes

```
class Mother(object):  
    ...  
class Father(object):  
    ...  
class Child(Mother, Father):  
    ...
```

- The new class inherits features from both parents
- But there are some rather tricky details
- Don't do it unless you know what you're doing

Exercise 4.2

Time : 30 Minutes

Special Methods

- Classes may define special methods
- Have special meaning to Python interpreter
- Always preceded/followed by __

```
class Stock(object):
    def __init__(self):
        ...
        ...

    def __repr__(self):
        ...
```

- There are several dozen special methods
- Will show a few examples

String Conversions

- Objects have two string representations

```
>>> from datetime import date  
>>> d = date(2012, 12, 21)  
>>> print(d)  
2012-12-21  
>>> d  
datetime.date(2012, 12, 21)  
>>>
```

- `str(x)` - Printable output

```
>>> str(d)  
'2012-12-21'  
>>>
```

- `repr(x)` - For programmers

```
>>> repr(d)  
'datetime.date(2012, 12, 21)'  
>>>
```

String Conversions

```
class Date(object):
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def __str__(self):
        return f'{self.year}-{self.month}-{self.day}'

    def __repr__(self):
        return f'Date({self.year},{self.month},{self.day})'
```

Note: The convention for `__repr__()` is to return a string that, when fed to `eval()`, will recreate the underlying object. If this is not possible, some kind of easily readable representation is used instead.

Methods: Mathematics

- Mathematical operators

a + b	a.__add__(b)
a - b	a.__sub__(b)
a * b	a.__mul__(b)
a / b	a.__div__(b)
a // b	a.__floordiv__(b)
a % b	a.__mod__(b)
a << b	a.__lshift__(b)
a >> b	a.__rshift__(b)
a & b	a.__and__(b)
a b	a.__or__(b)
a ^ b	a.__xor__(b)
a ** b	a.__pow__(b)
-a	a.__neg__()
~a	a.__invert__()
abs(a)	a.__abs__()

- Consult reference for further details

Methods: Item Access

- Methods used to implement containers

<code>len(x)</code>	<code>x.__len__()</code>
<code>x[a]</code>	<code>x.__getitem__(a)</code>
<code>x[a] = v</code>	<code>x.__setitem__(a,v)</code>
<code>del x[a]</code>	<code>x.__delitem__(a)</code>

- Use in a class

```
class Sequence(object):  
    def __len__(self):  
        ...  
    def __getitem__(self,a):  
        ...  
    def __setitem__(self,a,v):  
        ...  
    def __delitem__(self,a):  
        ...
```

Odds and Ends

- Defining new exceptions
- Bound and unbound methods
- Alternative attribute lookup

Defining Exceptions

- User-defined exceptions are defined by classes

```
class NetworkError(Exception):  
    pass
```

- Exceptions always inherit from Exception
- Usually, it's an empty class (use pass)
- You can also make a hierarchy

```
class AuthenticationError(NetworkError):  
    pass
```

```
class ProtocolError(NetworkError):  
    pass
```

Method Invocation

- Invoking a method is a two-step process
- Lookup: The `.` operator
- Method call: The `()` operator

```
class Stock(object):  
    ...  
    def cost(self):  
        return self.shares * self.price  
  
>>> s = Stock('GOOG', 100, 490.10)  
>>> c = s.cost ←————— Lookup  
>>> c  
<bound method Stock.cost of <Stock object at 0x590d0>>  
>>> c()  
49010.0 ←————— Method call  
>>>
```

Bound Methods

- A method that has not yet been invoked by the function call operator () is known as a "bound method"
- It operates on the instance where it originated

```
>>> s = Stock('GOOG', 100, 490.10)
>>> s
<Stock object at 0x590d0>
>>> c = s.cost
>>> c
<bound method Stock.cost of <Stock object at 0x590d0>>
>>> c()
49010.0
>>>
```

The diagram illustrates the binding of a method to an instance. An arrow labeled "binding" points from the variable `c` in the second line of the code to the method `cost` in the third line. Another arrow points from the string "`<Stock object at 0x590d0>`" in the fourth line to the same `cost` method, indicating that the method is bound to this specific instance.

Bound Methods

- Why would you care?
- Often a source of careless non-obvious errors

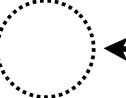
```
>>> s = Stock('GOOG', 100, 490.10)
>>> print('Cost : %0.2f' % s.cost)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: float argument required
>>>
```



Note missing ()

- Or devious behavior that's hard to debug

```
f = open(filename, 'w')
...
f.close
```



Oops. Didn't do anything at all

Attribute Access

- These functions may be used to manipulate attributes given an attribute name string

```
getattr(obj, 'name')           # Same as obj.name
setattr(obj, 'name', value)    # Same as obj.name = value
delattr(obj, 'name')          # Same as del obj.name
hasattr(obj, 'name')           # Tests if attribute exists
```

- Example: Probing for an optional attribute

```
if hasattr(obj, 'x'):
    x = getattr(obj, 'x'):
else:
    x = None
```

- Note: `getattr()` has a useful default value arg

```
x = getattr(obj, 'x', None)
```

Summary

- A high-level overview of classes
- Most code involving classes will involve the topics covered in this section
- If you're merely using existing libraries, the code is typically fairly simple

Exercise 4.3

Time : 15 Minutes

Section 5

The Inner Workings of Python Objects

Overview

- A few more details about how objects work
- How objects are represented
- Details of attribute access
- Data encapsulation techniques

Dictionaries Revisited

- A dictionary is a collection of named values

```
stock = {  
    'name' : 'GOOG',  
    'shares' : 100,  
    'price' : 490.1  
}
```

- Dictionaries are commonly used for simple data structures (shown above)
- However, they are used for critical parts of the interpreter and may be the most important type of data in Python

Dicts and Modules

- In a module, a dictionary holds all of the global variables and functions

```
# foo.py  
  
x = 42  
def bar():  
    ...  
def spam():  
    ...
```

foo.__dict__ or globals()

```
{  
    'x'      : 42,  
    'bar'    : <function bar>,  
    'spam'   : <function spam>  
}
```

Dicts and Objects

- User-defined objects also use dictionaries
 - Instance data
 - Class members
- In fact, the entire object system is mostly an extra layer that's put on top of dictionaries
- Let's take a look...

Dicts and Instances

- A dictionary holds instance data (`__dict__`)

```
>>> s = Stock('GOOG', 100, 490.1)
>>> s.__dict__
{'name': 'GOOG', 'shares': 100, 'price': 490.1}
```

- You populate this dict when assigning to `self`

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

`self.__dict__` →

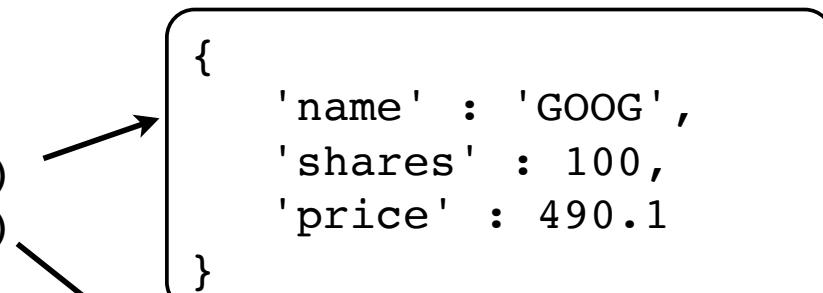
```
{
    'name' : 'GOOG',
    'shares' : 100,
    'price' : 490.1
}
```

instance data

Dicts and Instances

- Critical point : Each instance gets its own private dictionary

```
s = Stock('GOOG',100,490.1)  
t = Stock('AAPL',50,123.45)
```

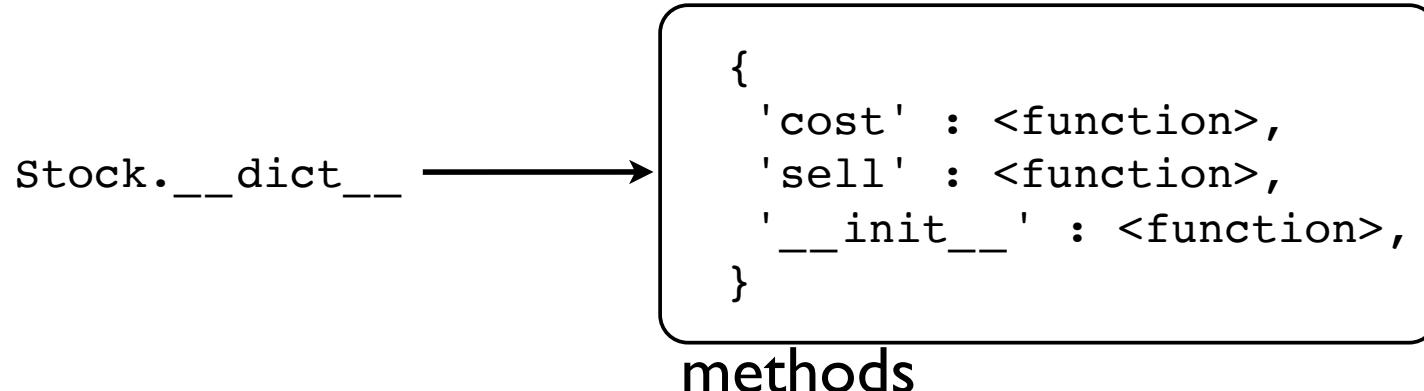


- So, if you created 100 instances of some class, there are 100 dictionaries sitting around holding data

Dicts and Classes

- A separate dictionary also holds the

```
class Stock(object):  
    def __init__(self, name, shares, price):  
        self.name = name  
        self.shares = shares  
        self.price = price  
    def cost(self):  
        return self.shares * self.price  
    def sell(self, nshares):  
        self.shares -= nshares
```



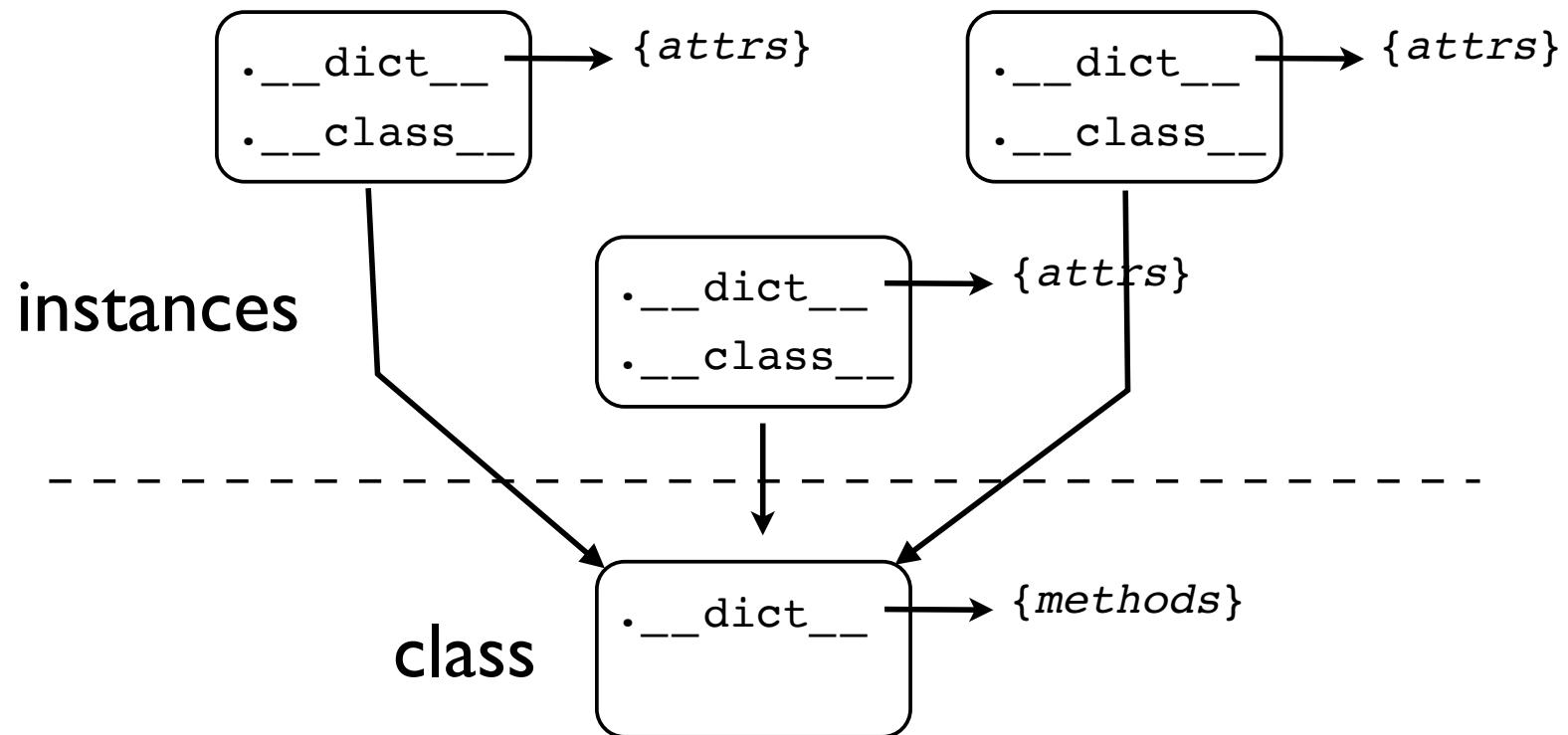
Instances and Classes

- Instances and classes are linked together
- `__class__` attribute refers back to the class

```
>>> s = Stock('GOOG',100,490.1)
>>> s.__dict__
{'name':'GOOG','shares':100,'price':490.1 }
>>> s.__class__
<class '__main__.Stock'>
>>>
```

- The instance dictionary holds data unique to each instance whereas the class dictionary holds data collectively shared by all instances

Instances and Classes



Attribute Access

- When you work with objects, you access data and methods using the (.) operator

```
x = obj.name      # Getting  
obj.name = value # Setting  
del obj.name     # Deleting
```

- These operations are directly tied to the dictionaries sitting underneath the covers

Modifying Instances

- Operations that modify an object update the underlying dictionary

```
>>> s = Stock('GOOG',100,490.1)
>>> s.__dict__
{'name': 'GOOG', 'shares': 100, 'price': 490.1}
→ >>> s.shares = 50
→ >>> s.date = '6/7/2007'
>>> s.__dict__
{'name': 'GOOG', 'shares': 50, 'price': 490.1,
 'date': '6/7/2007'}
→ >>> del s.shares
>>> s.__dict__
{'name': 'GOOG', 'price': 490.1, 'date': '6/7/2007'}
>>>
```

Reading Attributes

- Suppose you read an attribute on an instance

```
x = obj.name
```

- Attribute may exist in two places
 - Local instance dictionary
 - Class dictionary
- So, both dictionaries may be checked

Reading Attributes

- First check in local `__dict__`
- If not found, look in `__dict__` of class

```
>>> s = Stock(...)  s
>>> s.name
'GOOG'
>>> s.cost()
49010.0
>>>
```

The diagram illustrates the attribute lookup process. On the left, Python code creates an instance `s` of the `Stock` class and prints its `.name` and `.cost()` method. On the right, the state of the objects is shown. `s` is an instance of `Stock`, represented by a box containing `.__dict__` and `.__class__`. An arrow labeled '1' points from this box to a separate box containing the dictionary `{'name': 'GOOG', 'shares': 100}`. A downward arrow points from the `.__class__` field of `s` to the `Stock` class itself. The `Stock` class is represented by a box containing `.__dict__`. An arrow labeled '2' points from this box to a separate box containing the dictionary `{'cost': <func>, 'sell': <func>, '__init__': ...}`.

- This lookup scheme is how the members of a class get shared by all instances

How Inheritance Works

- Classes may inherit from other classes

```
class A(B,C):  
    ...
```

- Bases are stored as a tuple in each class

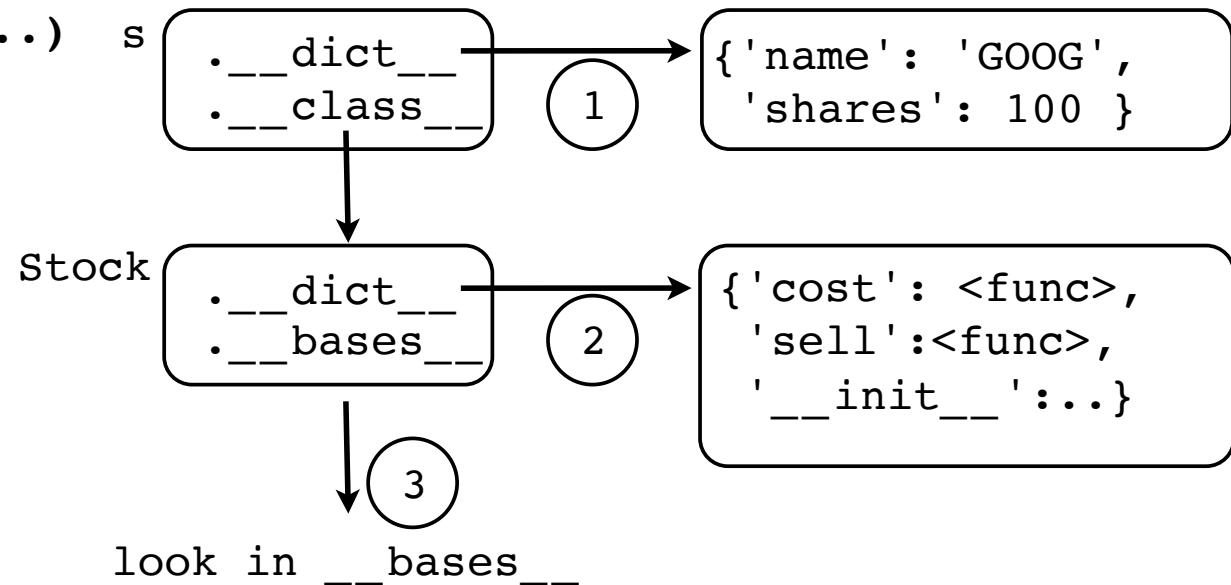
```
>>> A.__bases__  
(<class '__main__.B'>,<class '__main__.C'>)  
>>>
```

- This provides a link to parent classes

Reading Attributes

- First check in local `__dict__`
- If not found, look in `__dict__` of class
- If not found in class, look in base classes

```
>>> s = Stock(...)  
>>> s.name  
'GOOG'  
>>> s.cost()  
49010.0  
>>>
```

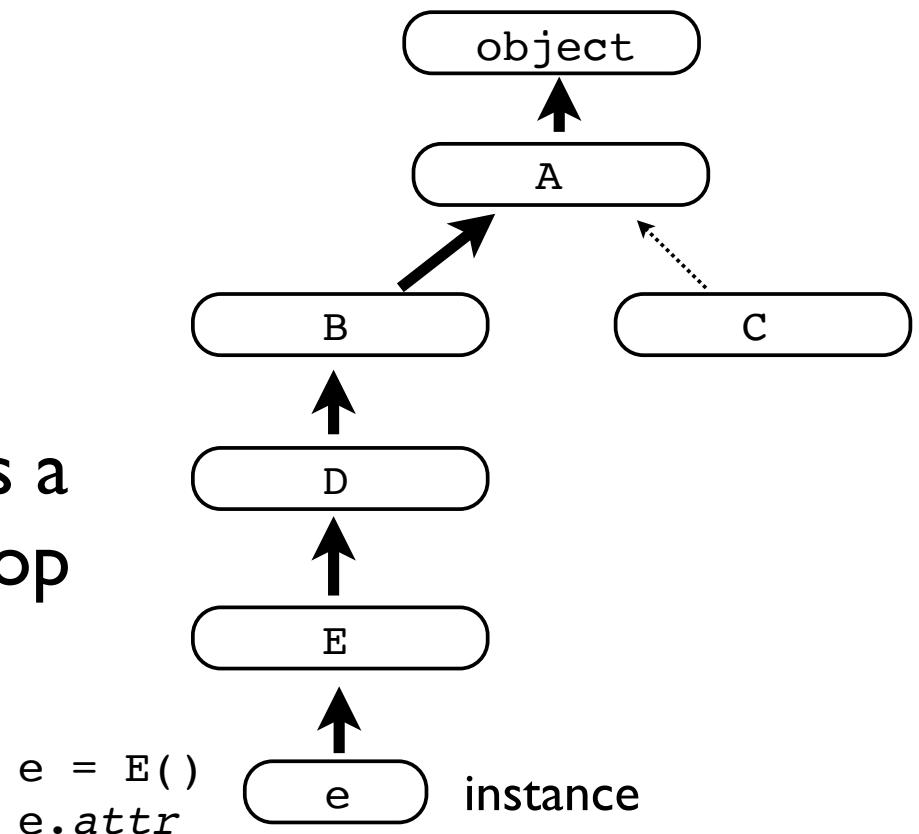


Single Inheritance

- In inheritance hierarchies, attributes are found by walking up the inheritance tree

```
class A(object): pass  
class B(A): pass  
class C(A): pass  
class D(B): pass  
class E(D): pass
```

- With single inheritance, there is a single path to the top
- You stop with the first match



The MRO

- The inheritance chain is precomputed and stored in the "MRO" attribute on the class

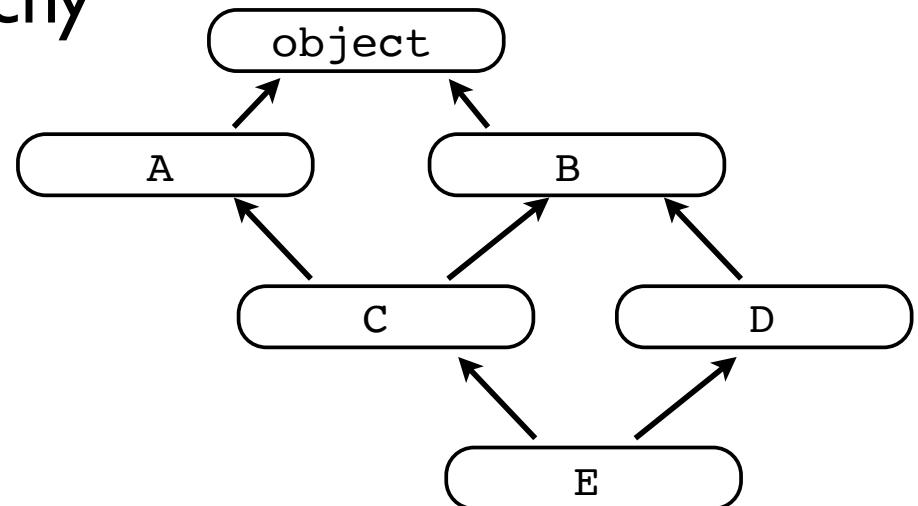
```
>>> E.__mro__
(<class '__main__.E'>,  <class '__main__.D'>,
 <class '__main__.B'>,  <class '__main__.A'>,
 <type 'object'>)
>>>
```

- "Method Resolution Order"
- To find attributes, Python walks the MRO
- First match wins

Multiple Inheritance

- Consider this hierarchy

```
class A(object): pass  
class B(object): pass  
class C(A, B): pass  
class D(B): pass  
class E(C, D): pass
```



- What happens here?

```
e = E()  
e.attr
```

- A similar search process is carried out, but what is the order? That is a problem.

Multiple Inheritance

- Python uses "cooperative multiple inheritance"
- There are some rules...

Rule 1: Children before parents
Rule 2: Parents go in order

- MRO is computed using those rules

```
>>> E.__mro__
(<class 'E'>, <class 'C'>, <class 'A'>, <class 'D'>,
 <class 'B'>, <class 'object'>)
>>>
```

The "Mixin" Pattern

```
class Base(object):
    def yow(self):
        print('Base.yow')

class A(Base):
    def yow(self):
        print('A.yow')
        super().yow()

class B(Base):
    def yow(self):
        print('B.yow')
        super().yow()

class C(A, B):
    pass
```

- Common parent
- A "wrapped" method
- Use of super()
- Sometimes used as a code composition tool

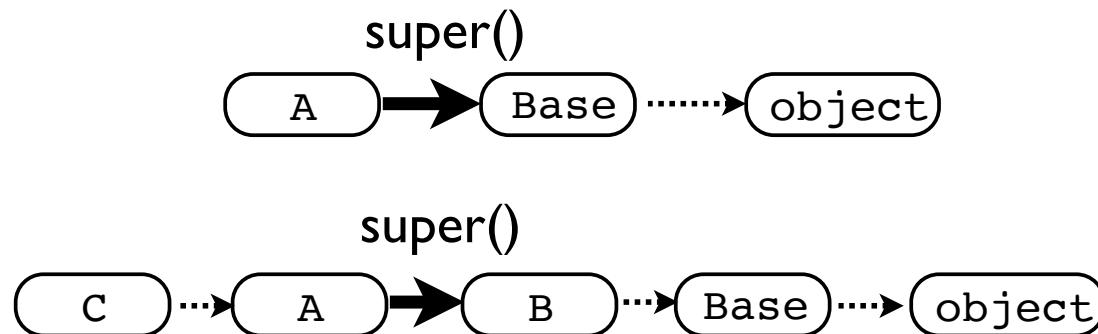
```
>>> c = C()
>>> c.yow()
A.yow
B.yow
Base.yow
>>>
```

Why super()?

- Always use super() when overriding methods

```
class A(Base):  
    def yow(self):  
        ...  
        return super().yow()
```

- super() delegates to the next class on the MRO



- Tricky bit: You don't know what it is

Some Cautions

- Multiple inheritance is a powerful tool
- With power comes responsibility
- Frameworks/libraries sometimes use it for advanced features involving composition of components
- More details in an advanced course

Exercise 5. I

Time : 10 Minutes

Classes and Encapsulation

- One of the primary roles of a class is to encapsulate data and internal implementation details of an object
- However, a class also defines a "public" interface that the outside world is supposed to use to manipulate the object
- This distinction between implementation details and the public interface is important

A Problem

- In Python, almost everything about classes and objects is "open"
 - You can easily inspect object internals
 - You can change things at will
 - There's no strong notion of access-control (i.e., private class members)
- If you're trying to isolate details of the internal "implementation" it's an issue

Python Encapsulation

- Python relies on programming conventions to indicate the intended use of something
- It's based on naming
- There is a general attitude that it is up to the programmer to observe the rules as opposed to having the language enforce rules

Private Attributes

- Any attribute name with a leading `_` is considered to be "private"

```
class Person(object):  
    def __init__(self, name):  
        self._name = 0
```

- However, it's only a programming style
- You can still access

```
>>> p = Person('Guido')  
>>> p._name  
'Guido'  
>>> p._name = 'Dave'  
>>>
```

Problem: Simple Attributes

- Consider the following class

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

s = Stock('GOOG', 100, 490.1)
s.shares = 50
```

- Suppose you later wanted to add validation

```
s.shares = '50'      # --> TypeError
```

- How would you do it?

Managed Attributes

- You might introduce accessor methods

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.set_shares(shares)
        self.price = price

    def get_shares(self): ←
        return self._shares

    def set_shares(self, value):
        if not isinstance(value, int):
            raise TypeError('Expected an int')
        self._shares = value
```

functions that layer get/
set operations on top of
a private attribute

- Too bad this breaks all existing code

s.shares = 50 → s.set_shares(50)

Properties

- An alternative approach to accessor

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    @property
    def shares(self):
        return self._shares

    @shares.setter
    def shares(self, value):
        if not isinstance(value, int):
            raise TypeError('Expected int')
        self._shares = value
```

- The syntax is a little jarring at first

Properties

- Normal attribute access triggers the

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    @property
    def shares(self):
        return self._shares

    @shares.setter
    def shares(self, value):
        if not isinstance(value, int):
            raise TypeError('Expected int')
        self._shares = value
```

The code defines a `Stock` class with an `__init__` method and three instance variables: `name`, `shares`, and `price`. It also contains a `@property` decorator for the `shares` attribute, which returns the value of `_shares`. Additionally, it has a `@shares.setter` decorator, which takes a `value` parameter and ensures it is an integer before setting it to `_shares`.

```
>>> s = Stock(...)
>>> s.shares
100
>>> s.shares = 50
>>>
```

- No changes needed to other source code

Properties

- You don't change existing attribute access

```
class Stock(object):
    def __init__(self, name, shares, price):
        ...
        self.shares = shares
        ...
    @property
    def shares(self):
        return self._shares

    assignment → @shares.setter
    calls the setter
    def shares(self, value):
        if not isinstance(value, int):
            raise TypeError('Expected int')
        self._shares = value
```

- Common confusion: property vs private name

Properties

- Also useful for computed data attributes

```
class Stock(object):  
    def __init__(self, name, shares, price):  
        self.name = name  
        self.shares = shares  
        self.price = price  
    @property  
    def cost(self):  
        return self.shares * self.price
```

- Example:

```
>>> s = Stock('GOOG', 100, 490.1)  
>>> s.shares ←———— Instance Variable  
100  
>>> s.cost ←———— Computed Value  
49010.0  
>>>
```

Uniform Access

- The last example shows how to put a more uniform interface on an object. If you don't do this, an object might be confusing to use:

```
>>> s = Stock('GOOG', 100, 490.1)
>>> a = s.cost()          # Method
>>> b = s.shares         # Data attribute
>>>
```

- Why is the () required for the cost, but not for the shares?

Decorator Syntax

- The @ syntax is known as "decoration"
- Specifies a modifier that's applied to a function definition that immediately follows

```
class Stock(object):  
    ...  
    @property  
    def cost(self):  
        return self.shares * self.price  
    ...
```

- It's kind of like a macro. More details are found in Section 9 (Advanced Topics)

__slots__ Attribute

- You can restrict the set of attribute names

```
class Stock(object):
    __slots__ = ('name', '_shares', 'price')
    ...
```

- Produces errors for other attributes

```
>>> s = Stock('GOOG', 100, 490.1)
>>> s.price = 385.15
>>> s.prices = 410.2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'Stock' object has no attribute 'prices'
```

- Prevents errors, restricts usage of objects
(but is actually used for performance)

Commentary

- Don't go overboard with private attributes, properties, slots, etc.
- They serve a specific purpose and you may see them when reading other Python code
- Not necessary for most day-to-day coding

Exercise 5.2

Time : 10 Minutes

Section 6

Generators

Iteration

- A simple definition: Looping over items

```
a = [2,4,10,37,62]
# Iterate over a
for x in a:
    ...
```

- A very common pattern
- loops, list comprehensions, etc.
- Most programs do a huge amount of iteration

Iteration Everywhere

- Many different objects support iteration

```
a = 'hello'  
for c in a:          # Loop over characters in a  
    ...  
  
b = { 'name': 'Dave', 'password':'foo' }  
for k in b:          # Loop over keys in dictionary  
    ...  
  
c = [1,2,3,4]  
for i in c:          # Loop over items in a list/tuple  
    ...  
  
f = open('foo.txt')  
for x in f:          # Loop over lines in a file  
    ...
```

Iteration: Protocol

- An inside look at the for statement

```
for x in obj:  
    # statements
```

- Underneath the covers

```
_iter = obj.__iter__()          # Get iterator object  
while True:  
    try:  
        x = _iter.__next__() # Get next item  
    except StopIteration:      # No more items  
        break  
    # statements  
    ...
```

- Objects that work with the for-loop all implement this low-level iteration protocol

Iteration: Protocol

- Example: Manual iteration over a list

```
>>> x = [1,2,3]
>>> it = x.__iter__()
>>> it
<listiterator object at 0x590b0>
>>> it.__next__()
1
>>> it.__next__()
2
>>> it.__next__()
3
>>> it.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
>>>
```

Supporting Iteration

- Knowing about iteration is useful if you want to add it to your own objects
- Example: Custom containers

```
class Portfolio(object):  
    def __init__(self):  
        self.holdings = []  
    def __iter__(self):  
        return self.holdings.__iter__()  
    ...  
  
port = Portfolio()  
for s in port:  
    ...
```

Exercise 6.1

Time : 10 Minutes

Customizing Iteration

- Suppose you wanted to create your own custom iteration pattern
- Example: Counting down...

```
>>> for x in countdown(10):
...     print(x, end=' ')
...
10 9 8 7 6 5 4 3 2 1
>>>
```

- It turns out there is an easy way to do it

Generators

- A function that defines iteration

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

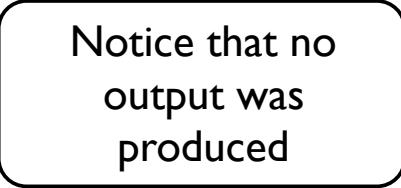
>>> for i in countdown(5):
...     print(i, end=' ')
...
5 4 3 2 1
>>>
```

- Any function that uses `yield` is a generator

Generator Functions

- Behavior is different than normal func
- Calling a generator function creates a generator object. It does not start running the function.

```
def countdown(n):
    print('Counting down from', n)
    while n > 0:
        yield n
        n -= 1
>>> x = countdown(10)←
>>> x
<generator object at 0x58490>
>>>
```



Notice that no output was produced

Generator Functions

- Function only executes on `__next__()`

```
>>> x = countdown(10)
>>> x
<generator object at 0x58490>
>>> x.__next__()
Counting down from 10
10
>>>
```

Function starts
executing here

- `yield` produces a value, but suspends function
- Function resumes on next call to `__next__()`

```
>>> x.__next__()
9
>>> x.__next__()
8
>>>
```

Generator Functions

- When the generator returns, iteration

```
>>> x.__next__()  
1  
>>> x.__next__()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in ?  
StopIteration  
>>>
```

- Observation :A generator function implements the same low-level protocol that the for statement uses on lists, tuples, dicts, files, etc.

Exercise 6.2

Time : 15 Minutes

Producers & Consumers

- Generators are closely related to various forms of "producer-consumer"

producer

```
def follow(f):
    ...
    while True:
        ...
        yield line
        ...
```

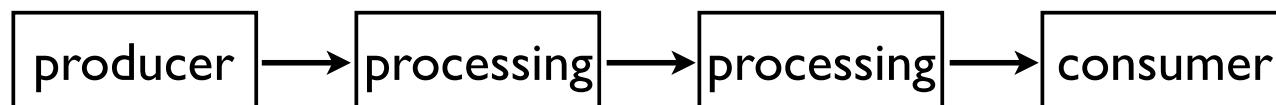
consumer

```
for line in follow(f):
    ...
```

- `yield` produces values
- `for` consume values

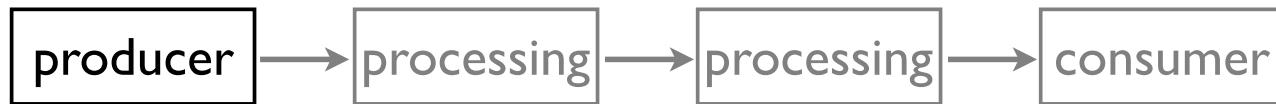
Generator Pipelines

- You can use this aspect of generators to set up processing pipelines (like Unix pipes)



- Processing pipes have an initial data producer, some set of intermediate processing stages, and a final consumer

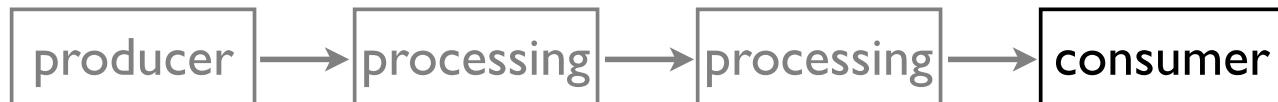
Generator Pipelines



```
def producer():
    ...
    yield item
    ...
```

- Producer is typically a generator (although it could also be a list or some other sequence)
- `yield` feeds data into the pipeline

Generator Pipelines

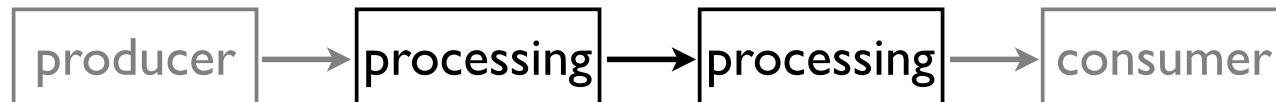


```
def producer():
    ...
    yield item
    ...
```

```
def consumer(s):
    for item in s:
        ...
    ...
```

- Consumer is a `for-loop`
- It gets items and does something with them

Generator Pipelines



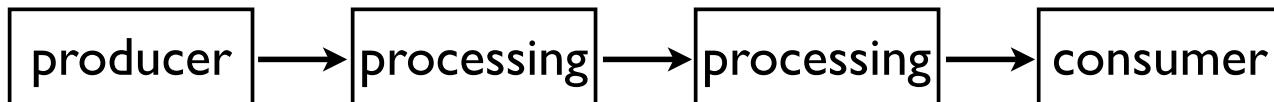
```
def producer():  
    ...  
    yield item  
    ...
```

```
def processing(s):  
    for item in s:  
        ...  
        yield newitem  
        ...
```

```
def consumer(s):  
    for item in s:  
        ...
```

- Intermediate processing stages simultaneously consume and produce items
- They might modify the data stream
- They can also filter (discarding items)

Generator Pipelines



```
def producer(): ...  
    yield item  
...  
  
def processing(s):  
    for item in s:  
        ...  
        yield newitem  
...  
  
def consumer(s):  
    for item in s:  
        ...
```

- Code to set up the pipeline

```
a = producer()  
b = processing(a)  
c = consumer(b)
```

- You will notice that data incrementally flows through the different functions

Exercise 6.3

Time : 15 minutes

Generator Expressions

- A generator version of a list comprehension

```
>>> a = [1,2,3,4]
>>> b = (2*x for x in a)
>>> b
<generator object at 0x58760>
>>> for i in b:
...     print(i, end=' ')
...
2 4 6 8
>>>
```

- Important differences
 - Does not construct a list.
 - Only useful purpose is iteration
 - Once consumed, can't be reused

Generator Expressions

- General syntax

(expression for i in s if conditional)

- Can also serve as a function argument

```
sum(x*x for x in a)
```

- Can be applied to any iterable

```
>>> a = [1,2,3,4]
>>> b = (x*x for x in a)
>>> c = (-x for x in b)
>>> for i in c:
...     print(i, end=' ')
...
-1 -4 -9 -16
>>>
```

Generator Expressions

- Main use of generator expressions is in code that performs some calculation on a sequence, but only uses the result once
- Example : Strip all comments from a file

```
f = open('somefile.txt')
lines = (line for line in f if not line.startswith('#'))
for line in lines:
    ...
f.close()
```

- Code runs faster and uses little memory (it's like a filter applied to a stream)

Why Use Generators?

- Many problems are much more clearly expressed in terms of iteration
- Looping over a collection of items and performing some kind of operation (searching, replacing, modifying, etc.)
- Processing pipelines can be applied to a wide range of data processing problems

Why Use Generators?

- Better memory efficiency
- Only produce values when needed
- Contrast to constructing giant lists
- Can operate on streaming data

Why Use Generators?

- Generators encourage code reuse
- Separates the "iteration" from code that uses the iteration.
- Can build a toolbox of interesting iteration functions and "mix-n-match"

The itertools Module

- A library module with various functions designed to help with iterators/generators

```
itertools.chain(s1,s2)
itertools.count(n)
itertools.cycle(s)
itertools.dropwhile(predicate, s)
itertools.groupby(s)
itertools.ifilter(predicate, s)
itertools imap(function, s1, ... sN)
itertools.repeat(s, n)
itertools.tee(s, ncopies)
itertools.izip(s1, ... , sN)
```

- All functions process data iteratively.
- Implement various kinds of iteration patterns

More Information

- "Generator Tricks for Systems Programmers" tutorial from PyCon'08

<http://www.dabeaz.com/generators>

- More examples and more generator tricks

Exercise 6.4

Time : 10 Minutes

Section 7

A Few Advanced Topics

Overview

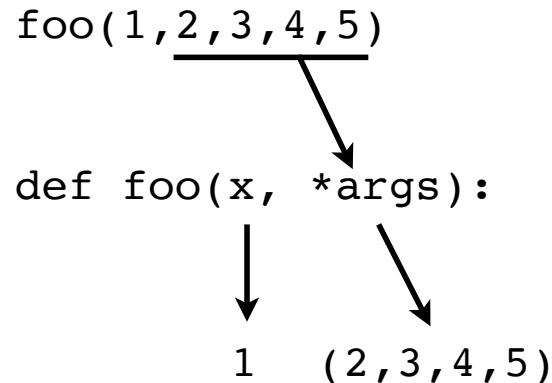
- More Python features you may encounter
 - Variable argument functions
 - Anonymous functions and lambda
 - Closures
 - Function decorators
 - Static and class methods

Variable Arguments

- Function that accepts any number of args

```
def foo(x, *args):  
    ...
```

- The arguments get passed as a tuple

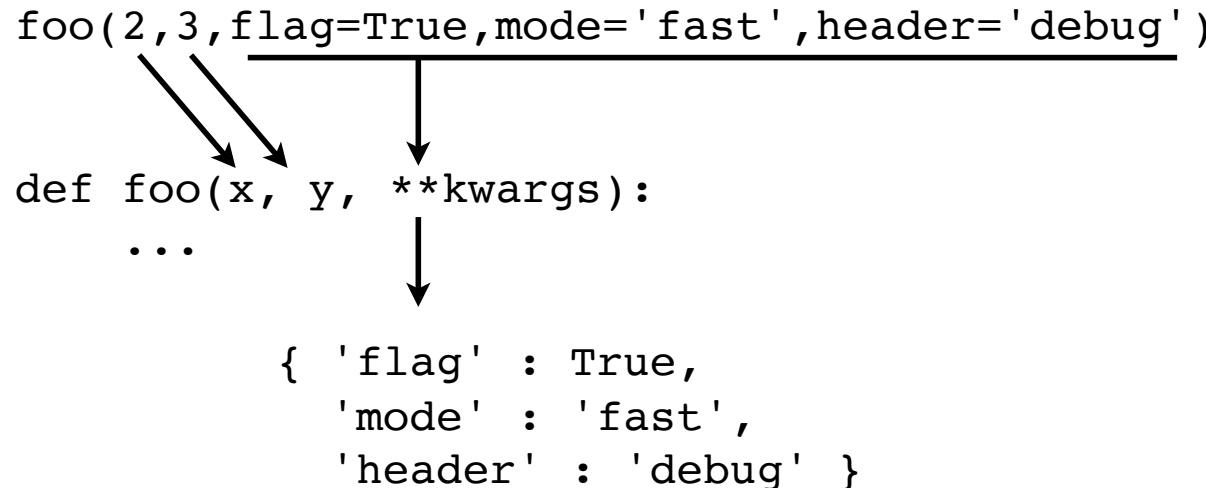


Variable Arguments

- Function that accepts any keyword args

```
def foo(x, y, **kwargs):  
    ...
```

- Extra keywords are passed in a dict



Variable Arguments

- A function that takes any arguments

```
def foo(*args, **kwargs):  
    statements
```

- Takes any combination of positional or keyword arguments
- Sometimes used when writing wrappers or when you want to pass arguments through to another function

Passing Tuples and Dicts

- Tuples can be expand into function args

```
args = (2,3,4)
foo(1, *args)      # Same as foo(1,2,3,4)
```

- Dictionaries can expand to keyword args

```
kwargs = {
    'color' : 'red',
    'delimiter' : ',',
    'width' : 400 }

foo(data, **kwargs)
# Same as foo(data,color='red',delimiter=',',width=400)
```

- These are not commonly used except when writing library functions.

Exercise 7.I

Time : 10 Minutes

List Sorting Revisited

- Lists can be sorted "in-place" (sort method)

```
s = [10,1,7,3]
s.sort()                      # s = [1,3,7,10]
```

- Sorting in reverse order

```
s = [10,1,7,3]
s.sort(reverse=True)  # s = [10,7,3,1]
```

- It seems "simple" enough...

List Sorting

- Sort this list of dicts

```
[{'name': 'AA', 'price': 32.2, 'shares': 100},  
 {'name': 'IBM', 'price': 91.1, 'shares': 50},  
 {'name': 'CAT', 'price': 83.44, 'shares': 150},  
 {'name': 'MSFT', 'price': 51.23, 'shares': 200},  
 {'name': 'GE', 'price': 40.37, 'shares': 95},  
 {'name': 'MSFT', 'price': 65.1, 'shares': 50},  
 {'name': 'IBM', 'price': 70.44, 'shares': 100}]
```

- Question: How?
- By what criteria? (name, shares, price)

List Sorting

- You can guide it using a "key function"

```
def stock_name(s):  
    return s['name']  
  
portfolio.sort(key=stock_name)
```

- Value returned by key func determines result

```
[{'name': 'AA', 'price': 32.2, 'shares': 100},  
 {'name': 'CAT', 'price': 83.44, 'shares': 150},  
 {'name': 'GE', 'price': 40.37, 'shares': 95},  
 {'name': 'IBM', 'price': 91.1, 'shares': 50},  
 {'name': 'IBM', 'price': 70.44, 'shares': 100},  
 {'name': 'MSFT', 'price': 51.23, 'shares': 200},  
 {'name': 'MSFT', 'price': 65.1, 'shares': 50}]
```

Callback Functions

- Callback functions are often short one-line functions that are only used for that one operation (e.g., sorting)
- Programmers often ask for a short-cut
- For example, is there some shorter way to specify the custom processing for `sort()`?

Anonymous Functions

- lambda expression

```
portfolio.sort(key=lambda s: s['name'])
```

- Creates an unnamed function that evaluates a single expression
- The above code is a shorter version of this

```
# Same as
def stock_name(s):
    return s['name']
```

```
portfolio.sort(key=stock_name)
```

Using lambda

- lambda is highly restricted
- Only a single expression is allowed
- No statements like "if", "while", etc.
- Most common use is with functions like sort()

Exercise 7.2

Time : 5 Minutes

Returning Functions

- Consider the following function

```
def add(x, y):  
    def do_add():  
        print('Adding', x, y)  
        return x + y  
    return do_add
```

- A function that returns another function?

```
>>> a = add(3,4)  
>>> a  
<function do_add at 0x6a670>  
>>> a()  
Adding 3 4  
7
```

- Notice that it works, but ponder it...

Local Variables

- Observe how the inner function refers to variables defined by the outer function

```
def add(x, y):  
    def do_add():  
        print('Adding', x, y)  
        return x + y  
    return do_add
```

- Further observe that those variables are somehow kept alive after add() has finished

```
>>> a = add(3,4)  
>>> a  
<function do_add at 0x6a670>  
>>> a()  
Adding 3 4  
7
```

Where are these values coming from?

Closures

- If an inner function is returned as a result, the inner function is known as a "closure"

```
def add(x, y):  
    def do_add():  
        print('Adding', x, y)  
        return x + y  
    return do_add
```

- Essential feature :A "closure" retains the values of all variables needed for the function to run properly later on

Using Closures

- Closures are an essential feature of Python
- However, their use is often subtle
- Common applications:
 - Use in callback functions
 - Delayed evaluation
 - Decorator functions (later)

Delayed Evaluation

- Consider a function like this:

```
def after(seconds, func):  
    time.sleep(seconds)  
    func()
```

- Example:

```
def greeting():  
    print('Hello Guido')  
  
after(30, greeting)
```

- It executes a supplied function... later

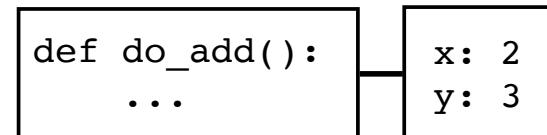
Delayed Evaluation

- Closures carry extra information around

```
def add(x, y):
    def do_add():
        print('Adding %s + %s -> %s' % (x, y, x + y))
    return do_add
```

```
def after(seconds, func):
    time.sleep(seconds)
    func()
```

```
after(30, add(2, 3))
```



- A function can have its own little environment

Code Repetition

- Closures can also be used as technique for avoiding excessive code repetition
- Almost like a macro or template
- You can write functions that make code

Exercise 7.3

Time : 15 Minutes

Function Decorators

- Closures are the basis for another Python feature known as "decoration"
- A topic related to "metaprogramming"
- An advanced topic that we'll only scratch the surface of right now

An Example

- Consider a function

```
def add(x, y):  
    return x + y
```

- Now, consider the function with some logging

```
def add(x, y):  
    print('Calling add')  
    return x + y
```

- Now, a second function with some logging

```
def sub(x, y):  
    print('Calling sub')  
    return x - y
```

- Observation: It's kind of repetitive

Observation

- Writing programs where there is a lot of code replication is usually really annoying
- Tedious to write
- Hard to maintain
- Especially if you decide that you want to change how it works (i.e., a different kind of logging perhaps).

An Example

- Perhaps you can make logging wrappers

```
def logged(func):  
    def wrapper(*args, **kwargs):  
        print('Calling', func.__name__)  
        return func(*args, **kwargs)  
    return wrapper
```

- Now, consider this bit of code

```
def add(x, y):  
    return x + y
```

```
add = logged(add)
```

- Question: What happens here?

```
add(3, 4)
```

An Example

- This example illustrates the process of creating a so-called "wrapper function"
- A wrapper is a function that wraps another function with some extra bits of processing

```
>>> add(3, 4)
Calling add
7
>>>
extra output
(added by the wrapper)
```

- Notice: the `logged()` function creates the wrapper and returns it as a result

Decorators

- Putting wrappers around functions is extremely common in Python

```
def add(x, y):  
    return x + y  
add = logged(add)
```

- So common, there is special syntax for doing it

```
@logged  
def add(x, y):  
    return x + y
```

- This performs the exact steps as shown at the top of the slide (it's just syntax)
- Is said to "decorate" the function

Commentary

- There are many more subtle details to decorators than what has been presented here
- For example, using them in classes
- Or using multiple decorators with a function
- However, the previous example is a good illustration of how their use tends to arise

Exercise 7.4

Time : 10 Minutes

Decorated Methods

- Predefined decorators are used to specify special kinds of methods in class definitions

```
class Foo(object):
    def bar(self,a):
        ...
    @staticmethod
    def spam(a):
        ...
    @classmethod
    def grok(cls,a):
        ...
    @property
    def name(self):
        ...
```

- Will briefly describe each one

Static Methods

- `@staticmethod` is used to define a so-called "static" class methods (from C++/Java)
- A function that's part of the class, but which does not operate on instances

```
class Foo(object):
    @staticmethod
    def bar(x):
        print('x =', x)

>>> Foo.bar(2)
x = 2
>>>
```

Using Static Methods

- Sometimes used to implement internal supporting code for a class
- Example : Code to help manage created instances (memory management, system resources, persistence, locking, etc.)

Class Methods

- `@classmethod` is used to define class methods
- A method that receives the `class` object as the first parameter instead of the instance

```
class Foo(object):
    def bar(self):
        print(self)
    @classmethod
    def spam(cls):
        print(cls)

>>> f = Foo()
>>> f.bar()
<__main__.Foo object at 0x971690> ← An instance
>>> Foo.spam()
<class '__main__.Foo'> ←————— A class
>>>
```

Using Class Methods

- Class methods are often used as a tool for defining alternate constructors

```
class Date(object):
    def __init__(self,year,month,day):
        self.year  = year
        self.month = month
        self.day   = day
    @classmethod
    def today(cls):
        tm = time.localtime()
        return cls(tm.tm_year, tm.tm_mon, tm.tm_mday)
d = Date.today()
```

Notice how the class passed
as an argument.



Using Class Methods

- Class methods solve some tricky problems with features like inheritance

```
class Date(object):  
    ...  
    @classmethod  
    def today(cls):  
        tm = time.localtime()  
        return cls(tm.tm_year, tm.tm_mon, tm.tm_mday)  
  
class NewDate(Date):  
    ...  
  
d = NewDate.today()
```

Gets the correct class
(e.g., NewDate)



Exercise 7.5

Time : 10 Minutes

Section 8

Testing and Debugging

Overview

- Testing
- Logging, error handling, and diagnostics
- Debugging
- Profiling

Testing Rocks, Debugging Sucks

- What else is there to say?
- Dynamic nature of Python makes testing critically important to most applications
- There is no compiler to find your bugs
- Only way to find bugs is to run the code and make sure you exercise all of its features

unittest Module

- You have some code

```
# simple.py

def add(x, y):
    return x + y
```

- Now create a separate file

```
# testsimple.py

import simple
import unittest
```

- Then define testing classes

```
class TestAdd(unittest.TestCase):
    ...
```

- Must inherit from unittest.TestCase

Using unittest

- In the class, define testing methods

```
class TestAdd(unittest.TestCase):  
    def test_simple(self):  
        # Test with simple integer arguments  
        r = simple.add(2, 2)  
        self.assertEqual(r, 5)  
  
    def test_str(self):  
        # Test with strings  
        r = simple.add('hello', 'world')  
        self.assertEqual(r, 'helloworld')
```

- Each method must start with "test..."

Using unittest

- Each test uses special assertions

```
# Assert that expr is True  
self.assertTrue(expr)
```

```
# Assert that x == y  
self.assertEqual(x,y)
```

```
# Assert that x != y  
self.assertNotEqual(x,y) # Assert x != y
```

```
# Assert that x is near y  
self.assertAlmostEqual(x,y,places)
```

```
# Assert that callable(arg1,arg2,...) raises exc  
self.assertRaises(exc,callable,arg1,arg2,...)
```

- There are others

Running unittests

- To run tests, turn the code into a script

```
# testsimple.py  
...  
if __name__ == '__main__':  
    unittest.main()
```

- Then run Python on the test file

```
bash % python3 testsimple.py  
F.  
=====  
FAIL: test_simple (__main__.TestAdd)  
-----  
Traceback (most recent call last):  
  File "testsimple.py", line 8, in test_simple  
    self.assertEqual(r, 5)  
AssertionError: 4 != 5  
-----  
Ran 2 tests in 0.000s  
FAILED (failures=1)
```

unittest comments

- There is an art to effective unit testing
- Can grow to be quite complicated for large applications
- The unittest module has a huge number of options related to test runners, collection of results, and other aspects of testing (consult documentation for details)

Third Party Test Tools

- pytest - A popular alternative

<https://pytest.org>

- coverage - Code coverage

<http://nedbatchelder.com/code/coverage/>

- mock - Mocking and testing library

<http://www.voidspace.org.uk/python/mock/>

Exercise 8.I

Time : 15 Minutes

logging Module

- A standard library module for recording diagnostic information
- It's also a very large module with a lot of sophisticated functionality
- Will show a simple example to illustrate

Exceptions Revisited

- In the exercises, we wrote a function `parse()` that looked something like this:

```
# fileparse.py
def parse(f, types=None, names=None, delimiter=None):
    records = []
    for line in f:
        line = line.strip()
        if not line: continue
        try:
            records.append(split(line,types,names,delimiter))
        except ValueError as e:
            print("Couldn't parse :", line)
            print("Reason :", e)
    return records
```

- Now, focus on the try-except section

Exceptions Revisited

- Do you print a warning message?

```
try:  
    records.append(split(line,types,names,delimiter))  
except ValueError as e:  
    print("Couldn't parse : ", line)  
    print("Reason : ", e)
```

- Or, do you silently ignore?

```
try:  
    records.append(split(line,types,names,delimiter))  
except ValueError as e:  
    pass
```

- Neither solution is satisfactory because you often want both behaviors (user selectable)

Using Logging

- The logging module can address this

```
# fileparse.py
import logging
log = logging.getLogger(__name__)

def parse(f,types=None,names=None,delimiter=None):
    ...
    try:
        records.append(split(line,types,names,delimiter))
    except ValueError as e:
        log.warning("Couldn't parse : %s", line)
        log.debug("Reason : %s", e)
```

- Here, code is modified to issue warning messages on a special "Logger" object

Logging Basics

- Creating a logger object

```
log = logging.getLogger(name)      # name is a string
```

- Issuing log messages

```
log.critical(message [, args])  
log.error(message [, args])  
log.warning(message [, args])  
log.info(message [, args ])  
log.debug(message [, args ])
```

each method
represents a different
level of severity

- Each of the above methods creates a formatted log message (*args* is used for % operator)

```
logmsg = message % args    # Written to the log
```

Logging Configuration

- Logging behavior is configured separately

```
# main.py
...
if __name__ == '__main__':
    import logging
    logging.basicConfig(
        filename = 'app.log',      # Log output file
        level    = logging.INFO,   # Output level
    )
```

- Typically, this is a one-time configuration at program startup
- Separate from code that makes logging calls

Big Picture

- Logging is highly configurable
- Can adjust every aspect of it (output files, levels, message formats, etc.)
- Code that uses logging doesn't have to worry about that however (it just issues messages)
- See: practical-python/Optional/Logging.pdf

Exercise 8.2

Time : 10 Minutes

Assertions

- assert statement

```
assert expr [, 'diagnostic message' ]
```

- If expression is not true, raises Assertion`Error` exception
- Should not be used to check user-input
- Useful for internal program checking

Contract Programming

- Consider assertions on all inputs and outputs

```
def add(x, y):  
    assert isinstance(x, int), 'Expected int'  
    assert isinstance(y, int), 'Expected int'  
    return x + y
```

- Checking inputs will immediately catch callers who aren't using appropriate arguments

```
>>> add(2, 3)  
5  
>>> add('2', '3')  
Traceback (most recent call last):  
...  
AssertionError: Expected int  
>>>
```

Optimized mode

- Python has an optimized run mode

```
bash % python3 -O prog.py
```

- This strips all assert statements
- Allows debug/release mode development
- Normal mode for full debugging
- Optimized mode for faster production runs

__debug__ variable

- Global variable checked for debugging

```
if __debug__:  
    # Perform some kind of debugging code  
    ...
```

- By default, __debug__ is True
- Set False in optimized mode (`python -O`)
- The implementation is efficient. The if statement is stripped in both cases and in `-O` mode, the debugging code is stripped entirely.

Error Handling

- Keeping Python alive upon termination

```
bash % python3 -i blah.py
Traceback (most recent call last):
  File "blah.py", line 13, in ?
    foo()
  File "blah.py", line 10, in foo
    bar()
  File "blah.py", line 7, in bar
    spam()
  File "blah.py", line 4, in spam
    x.append(3)
AttributeError: 'int' object has no attribute 'append'
>>>
```

- Python enters normal interactive mode
- Can use to examine global data, objects, etc.

The Python Debugger

- pdb module
- Entering the debugger after a crash

```
bash % python3 -i blah.py
Traceback (most recent call last):
  File "blah.py", line 13, in ?
    foo()
  File "blah.py", line 10, in foo
    bar()
  File "blah.py", line 7, in bar
    spam()
  File "blah.py", line 4, in spam
    x.append(3)
AttributeError: 'int' object has no attribute 'append'
>>> import pdb
>>> pdb.pm()
> /Users/beazley/Teaching/blah.py(4)spam()
-> x.append(3)
(Pdb)
```

The Python Debugger

- Launching the debugger inside a program

```
def some_function():
    statements
    ...
    import pdb; pdb.set_trace()      # Enter the debugger
    ...
    statements
```

- This starts the debugger at the point of the `set_trace()` call

Python Debugger

- Common debugger commands

(Pdb) help	# Get help
(Pdb) w(here)	# Print stack trace
(Pdb) d(own)	# Move down one stack level
(Pdb) u(p)	# Move up one stack level
(Pdb) b(reak) loc	# Set a breakpoint
(Pdb) s(tep)	# Execute one instruction
(Pdb) c(ontinue)	# Continue execution
(Pdb) l(ist)	# List source code
(Pdb) a(rgs)	# Print args of current function
(Pdb) !statement	# Execute statement

- For breakpoints, location is one of

(Pdb) b 45	# Line 45 in current file
(Pdb) b file.py:45	# Line 45 in file.py
(Pdb) b foo	# Function foo() in current file
(Pdb) b module.foo	# Function foo() in a module

Python Debugger

- Running entire program under debugger

```
bash % python3 -m pdb someprogram.py
```

- Automatically enters the debugger before the first statement (allowing you to set breakpoints and change the configuration)

Profiling

- cProfile module
- Collects statistics and prints a report
- Run run it from the command shell

```
bash % python3 -m cProfile someprogram.py
```

Profile Sample Output

```
bash % python3 -m cProfile cparse.py
        447981 function calls (446195 primitive calls) in
5.640 CPU seconds
```

Ordered by: standard name

	ncalls	tottime	percall	cumtime	percall	
filename:lineno(function)						
	2	0.000	0.000	0.000	0.000	:0(StringIO)
	101599	0.470	0.000	0.470	0.000	:0(append)
	56	0.000	0.000	0.000	0.000	:0(callable)
	4	0.000	0.000	0.000	0.000	:0(close)
	1028	0.010	0.000	0.010	0.000	:0(cmp)
	4	0.000	0.000	0.000	0.000	:0(compile)
	1	0.000	0.000	0.000	0.000	:0(digest)
	2	0.000	0.000	0.000	0.000	:0(exc_info)
	1	0.000	0.000	5.640	5.640	:0(execfile)
	4	0.000	0.000	0.000	0.000	:0(extend)
	50	0.000	0.000	0.000	0.000	:0(find)
	83102	0.430	0.000	0.430	0.000	:0(get)
...						

Summary

- Testing with unittest
- Logging
- Debugging (pdb)
- Profiling

Exercise 8.3

Time : 10 Minutes

Section 9

Packages

Overview

- Packages
- Third Party Modules

Modules (Reprise)

- Any Python source file is a module

```
# foo.py
def grok(a):
    ...
def spam(b):
    ...
```

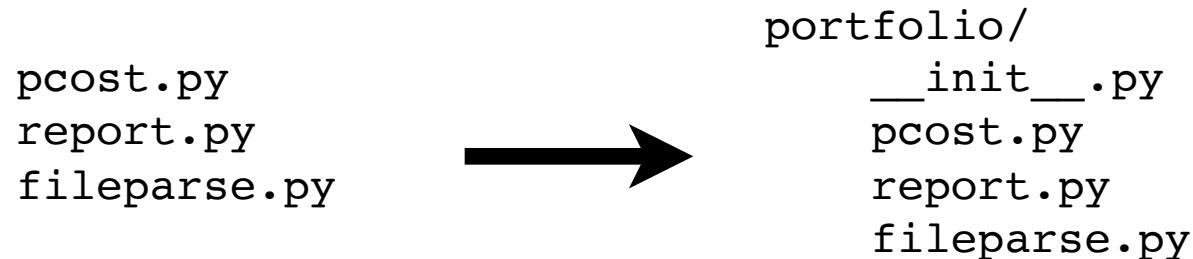
- import statement loads and executes a module

```
import foo

a = foo.grok(2)
b = foo.spam('Hello')
...  
...
```

Packages

- For larger collections of code, it is common to organize modules into a package



- Pick name and make a top-level directory
- Add an `__init__.py` file (may be empty)
- Put your source files into it

Using a Package

- Multilevel imports

```
import portfolio.report  
port = portfolio.report.read_portfolio('port.csv')
```

- Some variations

```
from portfolio import report  
port = report.read_portfolio('port.csv')
```

```
from portfolio.report import read_portfolio  
port = read_portfolio('port.csv')
```

Package-Relative Imports

- Imports between files in the same package require a slightly different form

```
portfolio/  
    __init__.py  
pcost.py  
report.py  
fileparse.py
```

```
# report.py  
from . import fileparse  
  
def read_portfolio(filename):  
    return fileparse.parse_csv(...)
```

- from .import modname**
- More explicitly indicates module location

`__init__.py` files

- Primary purpose of these files is to stitch modules together
- Example: Consolidating functions

```
# portfolio/__init__.py

from .pcost import portfolio_cost
from .report import portfolio_report
```

- Makes names appear at the "top level"

```
from portfolio import portfolio_cost
portfolio_cost('Data/portfolio.csv')
```

Package Scripts

- Running package submodules as scripts

```
portfolio/
    __init__.py
    pcost.py
    report.py
    fileparse.py
```

- `python -m package.module`

```
bash $ python3 -m portfolio.pcost Data/portfolio.csv
```

- Runs the code in the package environment

`__main__.py` file

- Package may optionally include `__main__.py`

```
portfolio/
    __init__.py
    __main__.py
    pcost.py
    report.py
    fileparse.py
```

- Specifies a package "entry point"
- Allows execution of the top-level package

```
bash $ python3 -m portfolio
```

Exercise 9.I

Time : 15 Minutes

Third Party Modules

- Python has a large library of built-in modules ("batteries included")
- There are even more third party modules
- Python Package Index (PyPi)

<http://pypi.python.org/>

- Or just do a Google search for a topic

Some Notable Modules

- requests : Accessing web services
- numpy, scipy : Arrays and vector mathematics
- pandas : Stats and data analysis
- django, flask : Web programming
- sqlalchemy : Databases and ORM
- ipython : Alternative interactive shell

Installing Modules

- Most common technique: pip

```
bash % python3 -m pip install packagename
```

- Downloads the package
- Installs it in the site-packages directory

```
/usr/local/lib/python3.6/site-packages
```

Manual Installation

- You downloaded a Python module or package using a standard file format such as a .gz, .tar.gz, .tgz, .bz2, or .zip file
- Unpack the file and look for a setup.py file in the resulting folder
- Run Python on that setup.py file

```
bash % python3 setup.py install
Installation messages
...
...
```

Commentary

- Installing third party modules is always a delicate matter
- More advanced modules may involve C/C++ code which has to be compiled to native code on your platform.
- May have dependencies on other modules
- In practice, it can be very difficult if you're building the entire environment from source

Exercise 9.2

(Optional)

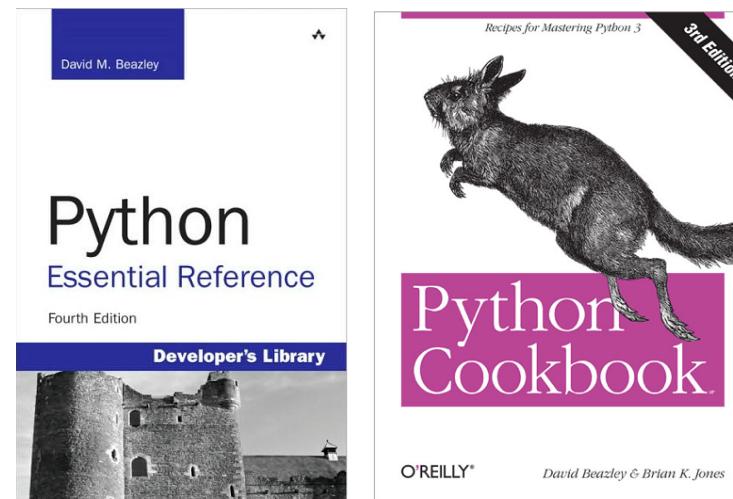
Time : 15 Minutes

That's All Folks!

- End of the course
- Material covered should be enough to get you going with most aspects of Python development
- Much of your work is now going to involve features of various libraries and packages
- Hope you enjoyed the class!

Shameless Plug

- Twitter: @dabeaz
- Buy my books



- Video: <https://www.safaribooksonline.com>



Python Programming Language

★★★★★ 50 reviews

by David Beazley

Publisher: Addison-Wesley Professional

Release Date: August 2016

ISBN: 9780134217314

Topics: Python

[View table of contents](#)