Advanced Python Mastery

David Beazley (@dabeaz)
https://www.dabeaz.com

Copyright (C) 2007-2023
(CC BY-SA 4.0), David Beazley

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Table of Contents

0. Course Setup
1. Python Review (Optional)
2. Idiomatic Data Handling
3. Classes and Objects
4. Inside Python Objects
5. Functions, errors, and testing
6. Working with Code
7. Metaprogramming
8. Iterators, Generators, and Coroutines
9. Modules and Packages

This course is an introduction to Python's more advanced features, with a focus on how they are applied in larger applications and frameworks. The target audience is software developers and anyone who wants to take their Python skills far beyond simple script writing. A major goal of the course is to understand how you can take control over the behavior of the language itself and bend it in ways that serve the needs of your application. By the end of this course, you'll know a lot more about the magic used by frameworks, different design options, and associated tradeoffs.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

About the Author

David Beazley has been writing Python since 1996 and is the author of "Python Distilled" (Addison-Wesley), "Python Cookbook, 3rd Edition" (O'Reilly Media), and the "Python Programming Language: LiveLessons" video series (Addison-Wesley). He regularly teaches advanced programming courses

https://www.dabeaz.com/courses.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

Copy

Course License and Usage

- Official course materials are here

https://github.com/dabeaz-course/python-mastery

This course is released under the Creative Commons Attribution-Share Alike 4.0 International (CC-SA 4.0) license.

You are free to use and adapt this material in any way that you wish as you as you give attribution to the original source. If you choose to use selected presentation slides in your own course, I kindly ask that the copyright notice, license, and authorship in the lower-left corner remain present.

About This Course

- Overheard:

  "The power of Python grows according to the skill and experience of the person using it."

- This course is about <u>that!</u>
- Going far beyond the tutorial
- Looking at things that you might care about if you're going to write "real" programs

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Target Audience

- Programmers who are writing Python code that will be used by others (libraries, application frameworks, domain-specific languages, etc.)
- Topics strongly focus on software development
  - Proper usage of Python features
  - Performance tradeoffs
  - Design options

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

System Requirements

- You should be using Python 3.6 or newer
- You need a local development environment
- Any operating system is fine

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Prerequisites

- This course assumes that you have already been using Python in some manner
- Previously completed some kind of online tutorial, taken an introductory course, or just worked on some code yourself.
- You should know the basics of running the interpreter and writing programs

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Section 0

Course Setup

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Required Files

- Where to get Python (if not installed)

  http://www.python.org

- This course is written for Python 3.6

- Exercises for this class

  github.com/dabeaz-course/python-mastery

- You should clone/fork the repo first

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Working Environment

- This is <u>not</u> an introductory course
- Use whatever tools you currently use to develop Python code
- Editors, IDEs, etc.
- Almost everything in this course is platform-neutral and will work everywhere

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Class Exercises

- Exercise descriptions are found in

  python-mastery/Exercises/

- All exercises have solution code

Look for the link at the bottom!

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Class Exercises

- Working solution code can be found in

  python-mastery/Solutions

- Each problem has its own directory

2_1/	Exercise 2.1
2_2/	Exercise 2.2
...	

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

General Tips

- Save all of your work in "python-mastery"
- Exercises assume this structure
- Exercises are meant to be worked in order
- If stuck, you can often start by copying solution code from Solutions/ and working forward.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Section 1

Python Review

(Optional)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Overview

- A very fast-paced review of Python
- The absolute basics that you should already know if you are taking this course
- Essential details for later parts of the class

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

All Things Python

http://www.python.org

- Downloads
- Documentation and tutorial
- Community Links
- News and more
- Tutorial

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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 27 2017, 13:20:23)
[GCC 4.2.1 Compatible Apple LLVM 6.1.0 (clang-602.0.53)]
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Creating Programs

- Programs are put in .py files

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

- Source files are simple text files
- Create with your favorite editor (e.g., emacs)
- Make sure you use "python" mode

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Running Programs

- Command line

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

- #! (Unix)

```
#!/usr/bin/env python3
# helloworld.py
print('hello world')
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

python -i

- For debugging, use python -i

```
bash % python3 -i helloworld.py
hello world
>>>
```

- Runs your program and then enters the Python interactive shell afterwards
- Quite useful for debugging, testing, etc.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 1.1

Time: 10 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Program Execution

- 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.
- When there are no more statements, the program stops

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Comments

- Comments are denoted by #

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

- Extend to the end of the line
- There are no block comments in Python (e.g., /* ... */).

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Variables

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

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

- Differs from C++/Java where variables have a fixed type that must be declared.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Expressions

- Math works normally (precedence, assoc, etc.)

```
a = 2 + 3
b = 2 + 3 * 4
c = (2 + 3) * 4
```

- Operators are same as C

```
+, -, *, /, %, <<, >>, &, |, ^, ...
```

- Other operators (Python-specific)

```
7 // 4              Truncating division
7 ** 4              Power operator
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Conditionals

- If-else

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

- If-elif-else

```
if a == '+':
    op = PLUS
elif a == '-':
    op = MINUS
elif a == '*':
    op = TIMES
else:
    op = UNKNOWN
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Relations

- 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')
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Looping

- While statement loops on a condition

```
while count > 0:
    print('T-minus', count)
    count -= 1
print('Boom!')
```

- For-loop iterates over items (e.g., foreach)

```
nums = [1,7,10,23]
for x in nums:
    print(x)          # Prints 1, 7, 10, 23
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Looping Control-Flow

- break - terminates a loop early

```
for name in names:
    if name == 'python':
        break
    ...
```

- continue - skip to next iteration

```
for line in lines:
    if line == '\n':   # Ignore blank lines
        continue
    ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Printing

- The print function

```
print(x)
print(x,y,z)
print('Your name is', name)
```

- Produces a single line of text
- Items are separated by spaces

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Formatted Printing

- f-strings

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

- .format() method

Come back to this

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

- Use % operator

```
print('%10s %10d %10.2f' % (name, shares, price))
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

pass statement

- Sometimes you will need to specify an empty block of code

```
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)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Core Python Objects

- Programs are built upon a core set of built-in datatypes (numbers, strings, lists, etc.)

```
None                    # Nothing, nada, nil, ...
True                    # Boolean
23                      # Integer
2.3                     # Float
2+3j                    # Complex
'Hello World'           # String (Unicode)
b'Hello'                # Byte string
('www.python.org',80)   # Tuple
[1,2,3,4]               # List
{'name':'IBM', ... }    # Dictionary
```

- You should have already used most of these types if you've written any Python at all

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Manipulating Objects

- Objects are manipulated by operators

```
a + b                    # Add
x = a[i]                 # Indexed lookup
y = a[i:j]               # Slicing
a[i] = val               # Indexed assignment
x in a                   # Containment
... many others ...
```

- Also manipulated by various methods

```
a.find('python')
a.split(',')
b.append(2)
...
```

- Available operators/methods depends on the object being manipulated

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 1.2

Time: 15 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

File Input and Output

- Opening a file

```
f = open('foo.txt','r')      # Open for reading
g = open('bar.txt','w')      # Open for writing
h = open('log.txt','a')      # Open for appending
```

- To read data

```
line = f.readline()          # Read a line of text
data = f.read([maxbytes])    # Read data
```

- To write text to a file

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

- Closing a file (when done)

```
f.close()
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Closing Files

- Files need to be closed after use

```
f = open('foo.txt','r')
# Use f
...
f.close()
```

- In modern Python, use 'with'

```
with open('foo.txt','r') as f:
    # Use f
    ...
# Automatically closed here
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Common Idioms

- Reading a file line-by-line

```
with open('foo.txt','r') as f:
    for line in f:
        # Process the line
        ...
```

- Reading an entire file into a string

```
with open('foo.txt','r') as f:
    data = f.read()
```

- Write to a file

```
with open('foo.txt','w') as f:
    f.write('some text\n')
    ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Text Data

- When reading text, Python 3 assumes unicode
- You might need to give an encoding

```
f = open('foo.txt','r',encoding='latin-1')
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Binary Data

- Make sure you use the right file modes if reading binary-encoded data

```
f = open('foo.dat','rb')
g = open('bar.bin','wb')
```

- Data read/written to binary-mode files uses normal 8-bit strings
- Will just have a lot of non-printing control characters, embedded nulls, etc. for non-text

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 1.3

Time: 10 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Simple Functions

- Use functions for code you want to reuse

```
def sumcount(n):
    total = 0
    while n > 0:
        total += n
        n -= 1
    return total
```

- Calling a function

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

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Simple Functions

- Functions behave in a sane manner
  - Inner variables have local scope
  - Things like recursion work fine
  - You can have default arguments
- Will say more about functions later, but you should already know how to define and use simple function definitions

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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'
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Catching Multiple Errors

- Can catch different kinds of exceptions

```
try:
    ...
except ValueError as e:
    ...
except TypeError as e:
    ...
```

- Alternatively, if handling is the same

```
try:
    ...
except (ValueError, TypeError) as e:
    ...
```

- Catching any exception (danger awaits)

```
try:
    ...
except Exception as e:
    ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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.)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 1.4

Time: 10 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Objects

- Python is an object-oriented language
- All of the basic data types (integers, strings, lists, etc.) are examples of "objects"
- Objects involve data and a set of "methods" that carry out various operations

```
a = 'Hello World'     # A string object
b = a.upper()         # A method applied to the string

items = [1,2,3]       # A list object
items.append(4)       # A method applied to the list
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Classes

- You can make your own objects

```
class Player:
   def __init__(self, x, y):
       self.x = x
       self.y = y
       self.health = 100

   def move(self, dx, dy):
       self.x += dx
       self.y += dy

   def damage(self, pts):
       self.health -= pts
```

- What is a class?
- It's all of the function definitions that implement the various methods

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Instances

- Created by calling the class as a function

```
>>> a = Player(2, 3)
>>> b = Player(10, 20)
>>>
```

- Each instance gets its own data

```
>>> a.x
2
>>> b.x
10
>>>
```

- Invoke the methods as follows

```
>>> a.move(1, 2)
>>> a.damage(10)
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

__init__ method

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

```
>>> a = Player(2, 3)

class Player:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.health = 100
```

newly created object

- Mostly, it just stores the data attributes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Methods

- Functions that operate on instances

```
class Player:
    ...
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
```

- The object is passed as first argument

```
>>> a.move(1, 2)

def move(self, dx, dy):
    ...
```

- 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 the "this" pointer in C++.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 1.5

Time: 10 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Modules

- Any Python source file is a module

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

- import statement loads and <u>executes</u> a module

```
import foo

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Namespaces

- A module is a collection of named values (i.e., it's said to be a "namespace")
- The names are simply 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)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Globals Revisited

- 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)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

import as statement

- Changes the local name of a module

```
# bar.py
import math as m

a = m.sin(x)
```

- Exactly the same as import except the module object is assigned a different name
- The new name only applies locally within the source file that did the import (other files can import using the standard name without any confusion)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

from module import

- Lifts selected symbols out of a module and puts them into local scope

```
# bar.py
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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

from module import *

- Takes <u>all symbols</u> from a module and places them into local scope

```
# bar.py
from math import *

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

- Useful if you are going to use a lot of functions from a module and it's annoying to specify the module prefix all of the time

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

from module import *

- You should almost never use it in practice because it leads to poor code readability
- Example:

```
from math import *
from random import *

...
r = gauss(0.0,1.0)       # In what module?
```

- Makes it very difficult to understand someone else's code if you need to locate the original definition of a library function

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Main Module

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

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

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

__main__ check

- It is standard practice for modules that can run as a main program 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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Locating Modules

- Modules are loaded from directories on a special module search path (sys.path)

```
>>> import sys
>>> sys.path
['',
 '/usr/local/lib/python36.zip',
 '/usr/local/lib/python3.6',
 '/usr/local/lib/python3.6/plat-darwin',
 '/usr/local/lib/python3.6/lib-dynload',
 '/usr/local/lib/python3.6/site-packages']
>>>
```

- Most ImportError exceptions are due to issues with the path and/or filenames

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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 3.6.1 (default, Mar 24 2017, 17:58:49)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.38)]
>>> import sys
>>> sys.path
['', '/project/foo/pyfiles', ..., ]
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Summary

- This has been an overview of basics
- If you've already been programming Python for awhile, you should already know this material
- Later sections go into much more depth

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 1.6

Time: 5 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Section 2

Data Handling

Core Topics

- Data structures
- Containers and collections
- Iteration
- Understanding the builtins
- Object model

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Data Structures

- Real programs must deal with 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)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Data Structures

- Some options
  - Tuple
  - Dictionary
  - Class instance
- Let's take a short tour

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Tuples

- A collection of values packed together

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

- Can use like an array

```
name = s[0]
cost = s[1] * s[2]
```

- Unpacking into separate variables

```
name, shares, price = s
```

- Immutable

```
s[1] = 75     # TypeError. No item assignment
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Dictionaries

- An unordered set of values indexed by "keys"

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

- Use the key name to access

```
name = s['name']
cost = s['shares'] * s['price']
```

- Modifications are allowed

```
s['shares'] = 75
s['date'] = '7/25/2015'
del s['name']
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

User-Defined Classes

- A simple data structure class

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

- This gives you the nice object syntax...

```
>>> s = Stock('GOOG', 100, 490.1)
>>> s.name
'GOOG'
>>> s.shares * s.price
49010.0
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Variations on Classes

- Class definitions have some variants
  - Slots - Saves memory
  - Dataclasses - Reduces coding
  - Named tuples - Immutability/tuple behavior
- Let's take a quick tour

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Classes and Slots

- For data structures, consider adding __slots__

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

- Slots is a performance optimization that is specifically aimed at data structures
- Greatly reduces the memory usage

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Dataclasses

```
from dataclasses import dataclass

@dataclass
class Stock:
    name : str
    shares : int
    price: float
```

- Possibly a convenience for reducing the amount of code that must be written
- Some useful methods get created automatically
- However, types are NOT enforced

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Named Tuples

- Another variant on class definition

```
import typing

class Stock(typing.NamedTuple):
    name: str
    shares: int
    price: float
```

- Alternate formulation (in older code)

```
from collections import namedtuple

Stock = namedtuple('Stock',
                   ['name', 'shares', 'price'])
```

- Main feature: immutability

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Named Tuples

- Named tuples have same features as tuples (immutability, unpacking, indexing, etc.)

```
>>> s = Stock('GOOG', 100, 490.1)
>>> s[0]
'GOOG'
>>> name, shares, price = s
>>> print('%10s %10d %10.2f' % s)
      GOOG        100     490.10
>>> isinstance(s, tuple)
True
>>> s.name = 'ACME'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 2.1

Time : 30 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Containers

- Programs often have to work many objects
  - Lists (ordered data)
  - Sets (unordered data, no duplicates)
  - Dictionaries (unordered key-value data)
- The choice depends on the problem

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Lists

- Use a list when the <u>order</u> of data matters
- Example: A list of tuples

```
portfolio = [
    ('GOOG', 100, 490.1),
    ('IBM', 50, 91.1),
    ('CAT', 150, 83.44)
]

portfolio[0] ──────→ ('GOOG', 100, 490.1)
portfolio[1] ──────→ ('IBM', 50, 91.1)
```

- Lists can be sorted and rearranged

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Sets

- A set is an <u>unordered</u> collection of unique items

```
a = {'IBM','AA','AAPL' }
```

- Sets eliminate duplicates

```
names = ['IBM','YHOO','IBM','CAT','MSFT','CAT','IBM']
unique_names = set(names)
```

- Sets are useful for membership tests

```
members = set()

members.add(item)       # Add an item
members.remove(item)    # Remove an item

if item in members:     # Test for membership
    ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Dicts

- Useful for indices and lookup tables

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

- Common use

```
p = prices['IBM']              # Value lookup
p = prices.get('AAPL', 0.0)    # Lookup with default value
prices['HPE'] = 37.42          # Assignment

if name in prices:             # Membership test
   ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Dicts, Composite Keys

- Use tuples for multi-part keys

```
prices = {
    ('ACME','2017-01-01') : 513.25,
    ('ACME','2017-01-02') : 512.10,
    ('ACME','2017-01-03') : 512.85,
    ('SPAM','2017-01-01') : 42.1,
    ('SPAM','2017-01-02') : 42.34,
    ('SPAM','2017-01-03') : 42.87,
}
```

- Usage:

```
p = prices['ACME', '2017-01-01']
prices['ACME','2017-01-04'] = 515.20
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Comprehensions

- List comprehension

  `[ expression for item in sequence if condition ]`

- Set comprehension

  `{ expression for item in sequence if condition }`

- Dict comprehension

  `{ key:value for item in sequence if condition }`

- These often simplify the creation of lists, sets, and dictionaries from existing data

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Comprehensions

- General syntax

```
[expression for item in sequence if condition]
```

- What it means

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

- Similar for set and dict comprehensions

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Comprehension Examples

- Collecting values from data

```
names = [s['name'] for s in portfolio]
```

- Unique values

```
unique_names = {s['name'] for s in portfolio}
```

- Performing database-like queries

```
results = [s for s in portfolio if s['price'] > 100
                                   and s['shares'] > 50 ]
```

- Quick mathematics over sequences

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Collections Module

- Provides some variations on common data structures (useful for specialized problems)
  - defaultdict
  - Counter
  - deque
  - ... and more

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

defaultdict

- Automatic initialization of missing dict keys

```
from collections import defaultdict

>>> d = defaultdict(list)
>>> d
defaultdict(<class 'list'>, {})
>>> d['x']
[]
>>> d
defaultdict(<class 'list'>, {'x': []})
>>>
```

- Allows combined operations

```
>>> d['y'].append(42)
>>> d['y']
[42]
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Counter

- A dictionary specialized for counting items

```
from collections import Counter

>>> totals = Counter()
>>> totals['IBM'] += 20
>>> totals['AA'] += 50
>>> totals['ACME'] += 75
>>> totals
Counter({'ACME': 75, 'AA': 50, 'IBM': 20})
>>>
```

- Has some other nice features (i.e., ranking)

```
>>> totals.most_common(2)
[('ACME', 75), ('AA', 50)]
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

deque

- Double-ended queue

```
from collections import deque

>>> q = deque()
>>> q.append(1)
>>> q.append(2)
>>> q.appendleft(3)
>>> q.appendleft(4)
>>> q
deque([4, 3, 1, 2])
>>> q.pop()
2
>>> q.popleft()
4
>>>
```

- More efficient than a list for queuing problems

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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)
        ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Multi-Search

- Problem: Search multiple places

```
techs = {
    'IBM': 91.23,
    'AAPL': 123.45,
    'HPE': 34.23
}
```

```
auto = {
    'GM': 23.45,
    'TM': 87.20,
    'F': 51.1,
    'TTA': 64.45,
}
```

- Solution: ChainMap

```
from collections import ChainMap
allprices = ChainMap(techs, auto)

>>> allrices['HPE']
34.23
>>> allprices['F']
51.1
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Commentary

- collections is a useful module to know
- Simplifies many common data handling problems
- If you're not using it, you're missing out

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 2.2

Time : 45 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Iteration

- The for-loop iterates over a sequence

```
>>> names = ['IBM', 'YHOO', 'AA', 'CAT' ]
>>> for name in names:
...     print(name)
...
IBM
YHOO
AA
CAT
>>>
```

- It seems simple enough...

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Iterating on Tuples

- Consider a list of tuples

```
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)
]
```

- Iteration with unpacking

```
for name, shares, price in portfolio:
    ...
```

- Iteration with a "throwaway" value (use _)

```
for name, _, price in portfolio:
    ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Iterating on Varying Records

- Consider a list of varying sized data structures

```
prices = [
    ['GOOG', 490.1, 485.25, 487.5 ],
    ['IBM', 91.5],
    ['HPE', 13.75, 12.1, 13.25, 14.2, 13.5 ],
    ['CAT', 52.5, 51.2]
]
```

- Wildcard unpacking (Python 3 only)

```
for name, *values in prices:
    print(name, values)
```

```
name     values
'GOOG'   [490.1, 485.25, 487.5 ]
'IBM'    [91.5]
'HPE'    [13.75, 12.1, 13.25, 14.2, 13.5 ]
'CAT'    [52.5, 51.2]
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

zip() function

- Iterate on multiple sequences in parallel

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

for colname, val in zip(columns, values):
    # Loops with colname='name'   val='GOOG'
    #            colname='shares' val=100
    #            colname='price'  val=490.1
    ...
```

- Common use: Making dictionaries

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

- Caution: Truncates to shortest input length

```
zip(['a','b','c'], [1,2]) ⟶ ('a', 1), ('b', 2)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Keeping a Running Count

- enumerate(*sequence* [, *start*])

```
names = ['IBM', 'YHOO', 'CAT' ]
for n, name in enumerate(names):
   # Loops with n=0 name='IBM'
   #            n=1 name='YHOO'
   #            n=2 name='CAT'
```

- Example: Line number tracking on a file

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Iterating on Integers

- 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
- Caution: range() is often a "code smell" for problems being solved the "hard way"

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Sequence Reductions

- sum(s), min(s), max(s)

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

- Boolean tests: any(s), all(s)

```
>>> s = [False, True, True, False]
>>> any(s)
True
>>> all(s)
False
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Unpacking Iterables

- Consider these iterables

```
a = (1, 2, 3)
b = [4, 5]
```

- Making lists and tuples (Python 3.5+)

```
c = [ *a, *b ]     # c = [1, 2, 3, 4, 5]    (list)
d = ( *a, *b )     # d = (1, 2, 3, 4, 5)    (tuple)
```

- It's subtle, but maybe better than using +

```
>>> c = a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "list") to tuple
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Unpacking Dictionaries

- Consider these dicts

```
a = { 'name': 'GOOG', 'shares': 100, 'price':490.1 }
b = { 'date': '6/11/2007', 'time': '9:45am' }
```

- Combining into a single dict (Python 3.5+)

```
c = { **a, **b }

>>> c
{ 'name': 'GOOG', 'shares':100, 'price': 490.1,
  'date': '6/11/2007,'time': '9:45am' }
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Argument Passing

- Iterables can expand to positional args

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

func(*a, *b)    #  func(1,2,3,4,5)
```

- Dictionaries can expand to keyword args

```
c = {'x': 1, 'y': 2 }

func(**c)   func(x=1, y=2)
```

- Combinations fine as long as positional go first

```
func(*a, **c)
func(*a, *b, **c)
func(0, *a, *b, 6, spam=37, **c)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Generator Expressions

- A variant of a list comprehension that produces the results via iteration
- Slightly different syntax (parentheses)

```
nums = [1,2,3,4]
squares = (x*x for x in nums)
```

- Use iteration to get the results

```
for n in squares:
    ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Generators

- A generator can only be consumed once
- Example:

```
>>> nums = [1,2,3,4]
>>> squares = (x*x for x in nums)
>>> for n in squares:
        print(n, end=' ')

1 4 9 16
>>> for n in squares:
        print(n, end=' ')
>>>
```

← notice no output (spent)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using Generators

- Generators are useful in contexts where the result is an intermediate step
- For example:

```
def sumsquares(nums):
    squares = (x*x for x in nums)
    total = sum(squares)
    return total
```

- Observe: squares is a temporary value--no need to make a list

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Generator Arguments

- Generators expressions are sometimes embedded as a function argument

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

print(','.join(str(x) for x in items))

if any(name.endswith('.py') for name in filenames):
    ...
```

- It acts as a filter/transform on an iterable

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Generator Functions

- A function that feeds iteration

```
def squares(nums):
    for x in nums:
        yield x*x      # Emit a value
```

- To get the results, use iteration

```
for n in squares:
    ...
```

- This a more general form that can be used if the iteration processing is more complicated

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 2.3

Time : 10 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Secrets of the Builtins

- Programmers use the built-in types without giving them much thought
- However, they have some subtle behavior that is worth knowing about
- Especially if you want to write better code

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

What is a Builtin?

- An object that's part of the Python interpreter
- Usually implemented entirely in C
- In some sense, the most primitive kind of object that you can use in a program

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Under the Hood

- All objects have an id, class and a reference count

- The id is the memory address
- The class is the "type"
- Reference count used for garbage collection

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Builtin Representation

- None (a singleton)

- float (64-bit double precision)

- int (arbitrary precision)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

String Representation

- Strings adapt to Unicode (size may vary)

```
>>> a = 'n'
>>> b = 'ñ'
>>> sys.getsizeof(a)
50
>>> sys.getsizeof(b)
74
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Memory Overhead

- There is inherent memory overhead
- Can investigate with sys.getsizeof()

```
>>> import sys
>>> a = 2
>>> sys.getsizeof(a)
28
>>> b = 2.5
>>> sys.getsizeof(b)
24
>>> c = '2.5'
>>> sys.getsizeof(c)
52
>>>
```

- A big part of memory use in earlier exercises

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Operation of the Builtins

- The builtin types operate according to predefined "protocols" (special methods)

```
>>> a = 2
>>> b = 3
>>> a + b
5
>>> a.__add__(b)      # Protocol
5
>>> c = 'hello'
>>> len(c)
5
>>> c.__len__()       # Protocol
5
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Object Protocols

- The object protocols are baked into the interpreter at a very low level (byte code)

```
>>> def f(x, y):
...     return x + y
...
>>> import dis
>>> dis.dis(f)
  2           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                1 (y)
              4 BINARY_ADD
              6 RETURN_VALUE
>>>
```

hardwired

```
def __add__(self, other):
    ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Making New "Builtins"

- Awareness of protocols allows you to make new objects that behave like the builtins
- Example: decimals, fractions

```
>>> from fractions import Fraction
>>> a = Fraction(2, 3)
>>> b = Fraction(1, 2)
>>> a + b
Fraction(7, 6)
>>> a > 0.5
True
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 2.4

Time : 30 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Container Representation

- Container objects only hold <u>references</u> (pointers) to their stored values

```
items = [a, b, c, d, e]          # A list
```

- All operations involving the container internals only manipulate the pointers (not the objects)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Over-allocation

- All mutable containers (lists, dicts, sets) tend to over-allocate memory so that there are always some free slots available

- This is a performance optimization
- Goal is to make appends, insertions fast

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Example : List Memory

- Example of list memory allocation

```
items = []
items.append(1)
items.append(2)
items.append(3)
items.append(4)

items.append(5)
items.append(6)
items.append(7)
```

- Extra space means that most append() operations are very fast (space is already available, no memory allocation required)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Container Growth

- Memory use of containers grows in proportion to the number of stored values
- Lists : Increase by ~12.5% when full
- Sets : Increases by factor 4 when 2/3 full
- Dicts : Increases by a factor 2 when 2/3 full

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Set/Dict Hashing

- Sets and dictionaries are based on hashing
- Keys are used to determine an integer "hashing value" (__hash__() method)

```
a = 'Python'
b = 'Guido'
c = 'Dave'

>>> a.__hash__()
-539294296
>>> b.__hash__()
1034194775
>>> c.__hash__()
2135385778
```

- Value used internally (implementation detail)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Key Restrictions

- Sets/dict keys restricted to "hashable" objects

```
>>> a = {'IBM','AA','AAPL'}
>>> b = {[1,2],[3,4]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
```

- This usually means you can only use strings, numbers, or tuples (no lists, dicts, sets, etc.)
- Requires __hash__() and __eq__() methods

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Dict Layout

- Hashing in a nutshell….

- There's a list of entries (preserves insertion order)

- A hashed index (holds entry positions)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Collision Resolution

- Hash index is perturbed until an open slot found

```
key='name'
h = key.__hash__() -> 15034981
i = h % isize -> 5
```

- Recurrence

```
i, h = perturb(i, h, size)

i = 7, 6, 1, 4, 5, 2, 3, 0, ...
```

- Every slot is tried eventually
- Works better if many open slots available

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Container Protocols

- Containers also work through "protocols"

```
>>> a = ['x','y','z']
>>> a[1]
'y'
>>> a.__getitem__(1)        # Protocol
'y'
>>> 'z' in a
True
>>> a.__contains__('z')     # Protocol
True
>>>
```

- You can make custom container objects by implementing the required methods
- Examples: numpy, Pandas, etc.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Container Taxonomy

- For new containers, consider collections.abc

```
Mapping, MutableMapping
Sequence, MutableSequence
Set, MutableSet
```

- Use these as a base class

```
class MyContainer(collections.abc.MutableMapping):
    ...
```

- Forces you to implement required methods

```
>>> c = MyContainer()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class MyContainer
with abstract methods __delitem__, __getitem__, __iter__,
__len__, __setitem__
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 2.5

Time : 25 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Understanding 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 <u>never make a copy</u> of the value being assigned
- All assignments are merely reference copies (or pointer copies if you prefer)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Assignment Example

- Consider this code fragment:

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

- A picture of the underlying memory

There is only one list object [1,2,3], but there are four different references to it.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Assignment Caution

- Modifying a value affects <u>all</u> 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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Call by Object

- Objects are never copied on function call

```
def func(items):
    items.append(42)

>>> a = [1,2,3]
>>> func(a)
>>> a
[1, 2, 3, 42]
>>>
```

- Mutations affect the original object
- Reminder: many objects are immutable

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Reassigning Names

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

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

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

- The name now refers to a different object

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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
>>>
```

- Every object also has an integer identifier

```
>>> id(a)
2774760
>>> id(b)
2774760
>>>
```

The object identifier is kind of like a pointer. If two names have the same id value, they're referring to the same object.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exploiting Immutability

- Immutable values can be safely shared

```
portfolio = [
  {'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}
]
```

- Sharing can save significant memory

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Shallow Copies

- Containers have methods for copying

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

- However, items are copied by reference

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

This inner list is still being shared

- Known as a "shallow copy"

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Example: Emulating Cases

A big conditional with many cases

```
if op == '+':
    r = add(x, y)
elif op == '-':
    r = sub(x, y):
elif op == '*':
    r = mul(x, y):
elif op == '/':
    r = div(x, y):
```

Reformulation using a dict of functions

```
ops = {
  '+' : add,
  '-' : sub,
  '*' : mul,
  '/' : div
}

r = ops[op](x,y)
```

- Key idea: Can make data structures from anything

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 2.6

Time : 25 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Section 3

Classes and Objects

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

When to use Objects?

- Object oriented programming is largely concerned with the modeling of "behavior."
- An "object" consists of some internal state, but more importantly, has methods that make it do various things.
- The methods give an object its personality

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

An Example

- Data

```
host = ('www.python.org', 80)
```

- Behavior

```
c = Connection('www.python.org',80)
c.open()
c.send(data)
c.recv()
c.close()
```

- Data and behavior are bound together

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

The class statement

- Use 'class' to define a new object

```
class Player:
   def __init__(self, x, y):
       self.x = x
       self.y = y
       self.health = 100

   def move(self, dx, dy):
       self.dx += dx
       self.dy += dy

   def damage(self, pts):
       self.health -= pts
```

- What is a class?
- Mostly, it's a set of functions that carry out various operations on so-called "instances"

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Instances

- Instances are the actual "objects" that you manipulate in your program
- Created by calling the class as a function

```
>>> a = Player(2, 3)
>>> b = Player(10, 20)
>>>
```

- Emphasize: The class statement is just the definition (it does nothing by itself)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Instance Data

- Each instance has its own local data

```
>>> a.x
2
>>> b.x
10
```

- This data is initialized by __init__()

```
class Player:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.health = 100
```

Any value stored on "self" is instance data

- There are no restrictions on the total number or type of attributes stored

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Instance Methods

- Functions applied to instances of an object

```
class Player:
    ...
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
```

- The object is always passed as first argument

```
>>> a.move(1, 2)

def move(self, dx, dy):
    ...
```

- 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."

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Attributes

- A word on terminology
- "Attribute" is anything accessed via (.)

```
>>> a.x              # Attribute of an instance
2

>>> Player.move          # Attribute of a class
<function Player.move at 0x10e7f6400>

>>> import math
>>> math.pi              # Attribute of a module
```

- Don't read too much into it

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

History Lesson

- Python classes were one of the last major features implemented in the language
- Design goals included no changes in syntax (other than the class statement itself) and no changes to function scoping rules
- Hence : Instance methods are normal function definitions that simply receive the instance as the first argument (self)
- That's it

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Class Scoping

- Caution: Classes do not define a scope

```
??? NameError


class Player:
    ...
    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def left(self, amt):
        move(-amt, 0)          # NO. Calls global move()
        self.move(-amt, 0)     # YES.
```

- If want to operate on an instance, you <u>always</u> have to refer to it explicitly (e.g., self)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 3.1

Time : 20 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Manipulating Instances

- There are only three operations on instances

```
obj.attr              # Get an attribute
obj.attr = value      # Set an attribute
del obj.attr          # Delete an attribute
```

- Attributes can be freely added and deleted after an instance is created

```
>>> s = Stock('ACME', 50, 91.1)
>>> s.shares
50
>>> s.date = '10/31/2017'   # Add an attribute
>>> del s.name              # Delete an attribute
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Attribute Access Functions

- 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: Output

```
attributes = [ 'name', 'shares', 'price']
for attr in attributes:
    print(attr, '=', getattr(obj, attr))
```

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

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Method Invocation

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

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

>>> s = Stock('ACME', 50, 91.1)
>>> c = s.cost
>>> c
<bound method Stock.cost of <Stock object at 0x590d0>>
>>> c()
4555.0
>>>
```

Lookup

Method call

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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('ACME', 50, 91.1)
>>> s
<Stock object at 0x590d0>
>>> c = s.cost
>>> c
<bound method Stock.cost of <Stock object at 0x590d0>>
>>> c()
4555.0
>>>
```

binding

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Bound Methods

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

```
>>> s = Stock('ACME', 50, 91.1)
>>> print('Cost : %0.2f' % s.cost)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: a float is required
>>>
```

Note missing ()

- Or devious behavior that's hard to debug

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

Oops. Didn't do anything at all

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 3.2

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

More on Class Definitions

- A class contains definitions that are <u>shared by all instances</u> of the class

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

    def cost(self):
        return self.shares * self.price
```

- Shared : Defined once, used by all instances
- Example : There is one cost() function that gets used by all instances created

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Class Variables

- Classes may also define variables
- Known as "class variables"

```
class SomeClass:
   debug = False
   def __init__(self, x):
       self.x = x
   ...
```

- There are two access routes

```
>>> SomeClass.debug
False
>>> s = SomeClass(42)
>>> s.debug
False
>>>
```

(On the class itself)

(On an instance of the class)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using Class Variables

- Often used for settings applied to all instances

```
class Date:
   datefmt = '{year}-{month}-{day}'
   def __init__(self, year, month, day):
       self.year = year
       self.month = month
       self.day = day
   def __str__(self):
        return self.datefmt.format(year=self.year,
                                   month=self.month,
                                   day=self.day)
```

- Possibly changed via inheritance

```
class USDate(Date):
   datefmt = '{month}/{day}/{year}'
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Class Methods

- A method that operates on the class itself

```
class SomeClass:
    @classmethod
    def yow(cls):
        print('SomeClass.yow', cls)
```

- It's invoked on the class, not an instance

```
>>> SomeClass.yow()
SomeClass.yow <class '__main__.SomeClass'>
>>>
```

- The <u>class</u> is passed as the first argument

```
SomeClass.yow()
```

```
@classmethod
def yow(cls):
    ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using Class Methods

- Class methods are often used as a tool for defining alternate initializers

```
class Date:
    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)
```

Notice how the class passed as an argument.

```
d = Date.today()
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using Class Methods

- Class methods solve some tricky problems with features like inheritance

```
class Date:
    ...
    @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)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Static Methods

- A function that's defined as part of a class, but does <u>not</u> operate on instances or the class

```
class SomeClass:
    @staticmethod
    def yow():
        print('SomeClass.yow')
```

- Example:

```
>>> SomeClass.yow()
SomeClass.yow
>>>
```

- Notice: There is no hidden self argument

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using Static Methods

- Uses vary:
  - Utility functions used by various methods
  - Instance management/tracking
  - Finalization, resource management
  - Certain design patterns
- Might improve code clarity--grouping related functionality together within a class

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 3.3

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Python Encapsulation

- Python relies on programming conventions to indicate the intended use of something
- Typically, this is 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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Private Attributes

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

```
class Base:
   def __init__(self, name):
       self._name = name
```

- However, this is only a programming style
- You can still access it

```
>>> b = Base('Guido')
>>> b._name
'Guido'
>>> b._name = 'Dave'
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Complication

- Are "private" attributes visible to subclasses?

```
class Base:
   def __init__(self, name):
       self._name = name

class Child(Base):
   def spam(self):
       print('Spam', self._name)
```

- As a general rule, this is accepted practice
- Subclasses often extend/enhance the functionality of the parent (may need attributes)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Avoiding Name Collisions

- Variant : Attribute names with two leading _

```
class Base:
   def __init__(self, name):
       self.__name = name
```

- Attribute is <u>not</u> visible in subclasses

```
class Child(Base):
    def spam(self):
        print('Spam', self.__name)  # AttributeError
```

- Implemented as a name mangling trick

```
>>> b = Base('Guido')
>>> b._Base__name
'Guido'
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Problem: Simple Attributes

- Consider the following class

```
class Stock:
    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?

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Managed Attributes

- You might introduce accessor methods

```
class Stock:
    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)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Properties

- An alternative approach to accessor methods

```
class Stock:
    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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Properties

- Normal attribute access triggers the methods

```
class Stock:
    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
```

get

set

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

- <u>No changes</u> needed to other source code

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Properties

- You don't change existing attribute access

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

assignment calls the setter

- Common confusion: property vs private name

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Properties

- Properties are also useful if you are creating objects where you want to have a very consistent programming interface
- Example : Computed data attributes

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

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Properties

- Example use:

```
>>> s = Stock('ACME', 50, 91.1)
>>> s.name
'ACME'
>>> s.cost
4555.0
>>>
```

Instance Variable

Computed Property

- Commentary : Notice how there is no obvious difference between the attributes as seen by the user of the object

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

__slots__ Attribute

- You can restrict the set of attribute names

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

- Produces errors for other attributes

```
>>> s = Stock('ACME', 50, 91.1)
>>> s.shares = 75
>>> s.share = 75
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'Stock' object has no attribute 'share'
```

- This is a performance optimization (uses less memory, runs faster)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

__slots__ Cautions

- slots should only be used sparingly
- Be aware that it's presence can cause strange interaction with other parts of Python that are related to objects
- Advice : Do <u>not</u> use it except with classes that are simply going to serve as simple data structures

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Commentary

- The features described so far cover virtually everything that you will see in most Python class definitions
- Essential pieces
  - Instance data (assignment in __init__)
  - Methods (instance, static, class)
  - Properties
  - Private attributes, __slots__

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 3.4

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Inheritance

- A tool for specializing existing objects

```
class Parent:
    ...

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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
- In a nutshell: Extending existing code

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Inheritance Example

- Adding a new method

```
class MyStock(Stock):
    def panic(self):
        self.sell(self.shares)

>>> s = MyStock('GOOG', 100, 490.1)
>>> s.sell(25)
>>> s.shares
75
>>> s.panic()
>>> s.shares
0
>>>
```

- You can give new capabilities to existing objects

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Inheritance Example

- Redefining a method

```
class MyStock(Stock):
    def cost(self):
        return 1.25 * self.shares * self.price

>>> s = MyStock('GOOG', 100, 490.1)
>>> s.cost()
61262.5
>>>
```

- The new method takes the place of the old one
- Other methods are unaffected

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Inheritance and Overriding

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

```
class Stock:
    ...
    def cost(self):
        return self.shares * self.price
    ...
class MyStock(Stock):
    def cost(self):
        actual_cost = super().cost()
        return 1.25 * actual_cost
```

- Use super() to call previous version
- Caution: Python 2 is different

```
actual_cost = super(MyStock, self).cost()
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Inheritance and __init__

- With inheritance, you must initialize parents

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

class MyStock(Stock):
    def __init__(self, name, shares, price, factor):
        super().__init__(name, shares, price)
        self.factor = factor
    def cost(self):
        return self.factor * super().cost()
```

- Again, you should use super() as shown

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

"is a" relationship

- Inheritance establishes a type relationship

```
class Stock:
    ...

class MyStock(Stock):
    ...

>>> s = MyStock('ACME', 50, 91.1)
>>> isinstance(s, Stock)
True
>>>
```

- Important: objects defined via inheritance are a special version of the parent (same capabilities)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

object base class

- Sometimes a class will use object as parent

```
class Stock(object):
    ...
```

- object is the parent of all objects in Python (even if you don't specify it in Python 3)
- Note: There is some historical baggage with Python 2 so it's common to see it in old code

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Multiple Inheritance

- You can specify multiple base classes

```
class Parent1:
    ...
class Parent2:
    ...
class Child(Parent1, Parent2):
    ...
```

- The new class inherits features from both parents
- But there are some really tricky details (later)
- Don't do it unless you understand it

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using Inheritance

- Inheritance is often used as a code customization/extensibility feature
- For example, certain parts of a framework might involve inheriting from an existing class and redefining a handful of methods
- Idea: you add bits and pieces to existing code to make it do custom processing

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 3.5

Time : 20 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Special Methods

- Classes can customize almost every aspect of their behavior
- This is done through special methods

```
class Point:
    def __init__(self, x, y):
        ...
    def __str__(self):
        ...
```

- There are dozens of these methods
- Instead of showing every possible customization, will show essential ones

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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)'
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

String Conversions

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

   def __str__(self):
       return '%d-%d-%d' % (self.year,
                            self.month,
                            self.day)

   def __repr__(self):
       return 'Date(%r,%r,%r)' % (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.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Methods: Item Access

- Methods used to implement containers

```
len(x)          x.__len__()
x[a]            x.__getitem__(a)
x[a] = v        x.__setitem__(a,v)
del x[a]        x.__delitem__(a)
a in x          x.__contains__(a)
```

- Definition in a class

```
class Container:
    def __len__(self):
        ...
    def __getitem__(self,a):
        ...
    def __setitem__(self,a,v):
        ...
    def __delitem__(self,a):
        ...
    def __contains__(self,a):
        ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Instance Creation

- Instances are created in two steps

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

d = Date(2012, 12, 21)
```

- Under the hood

```
d = Date.__new__(Date, 2012, 12, 21)
d.__init__(2012, 12, 21)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using __new__

- Sometimes you might use __new__() directly

```
class Date:
    ...
    @classmethod
    def today(cls):
        t = time.localtime()
        self = cls.__new__(cls)
        self.year = t.tm_year
        self.month = t.tm_mon
        self.day = t.tm_mday
        return self

d = Date.today()
```

- Creates an instance, but bypasses __init__()

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Defining __new__

- Classes may define __new__()

```
class A:
    @staticmethod
    def __new__(cls, x, y):
        ...
        return super().__new__(cls)
    def __init__(self, x, y):
        ...
```

- Not common, but sometimes used when altering some tricky aspect of instance creation
  - Instance caching
  - Immutability

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

__del__ method

- Classes might define a "destructor" method

```
class Connection:
    ...
    def __del__(self):
        # Cleanup statements
        ...
```

- Called when the reference count reaches 0
- Confusion: Not related to “del” operator

```
c = Connection()                # refcnt = 1
d = c                           # refcnt = 2

del d       # Doesn't call d.__del__()  (refcnt = 1)
c = None    # Calls c.__del__()  (refcnt = 0)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

__del__ method

- Typical uses:
  - Proper shutdown of system resources (e.g., network connections)
  - Releasing locks (e.g., threading)
- Avoid defining it for any other purpose

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Weak References

- A weak reference is a reference to an object that does <u>not</u> increase its reference count
- Sometimes this is desired in when there is a complicated relationship between objects and there are issues with memory management
- Supported by weakref library module

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

weakref module

- Creating a weak reference

```
>>> import weakref
>>> f = Foo()
>>> fref = weakref.ref(f)
>>> fref
<weakref at 0x4203c0; to 'Foo' at 0x41dff0>
```

- Getting the object being pointed at

```
>>> g = fref()          # Dereference
>>> print(g)
<__main__.Foo object at 0x41dff0>
```

- If object is dead, deference returns None

```
>>> g = fref()
>>> print(g)
None
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using Weak References

- Weak references are sometimes used where there are reference cycles between objects
- Example : graphs, trees, observers, caches,etc.
- Not something you should consider unless dealing with really tricky memory problems
- More features in the weakref module

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Context Managers

- For resources, consider the use of the 'with' statement instead of relying on __del__()

```
with obj as val:  ──────────→ val = obj.__enter__()
    statements
    statements
    statements
    ...
    statements
          ──────────────────→ obj.__exit__(ty, val, tb)
```

- Allows you to customize entry/exit steps

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Context Managers

- Example:

```
class Manager:
    def __enter__(self):
        print('Entering')
        return self
    def __exit__(self, ty, val, tb):
        print('Leaving')
        if type:
            print('An exception occurred')
```

Note: the ty, val, tb arguments have information about pending exceptions (if any)

- Example use:

```
>>> m = Manager()
>>> with m:
...     print('Hello World')
...
Entering
Hello World
Leaving
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 3.6

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Code Reuse

- A major theme of object oriented programming concerns code reuse and making things extensible
- A big topic
- There are a number of common techniques

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Interfaces

- Classes often serve as a kind of design specification or programming interface

```
class IStream:
    def read(self, maxbytes=None):
        raise NotImplementedError()
    def write(self, data):
        raise NotImplementedError()
```

- This class isn't used directly, but is usually included as a base class for other objects

```
class UnixPipe(IStream):
    def read(self, maxbytes=None):
        ...
    def write(self, data):
        ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Abstract Base Classes

- Consider defining interfaces as an abstract base class (ABC) instead

```
from abc import ABC, abstractmethod

class IStream(ABC):
    @abstractmethod
    def read(self, maxbytes=None):
        pass
    @abstractmethod
    def write(self, data):
        pass
```

- Doesn't allow instantiation unless all of the abstract methods have been fully implemented

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Abstract Base Classes

- ABCs may simplify type checking

```
def write_data(data, stream):
    if not isinstance(stream, IStream):
        raise TypeError('Expected a Stream')
    ...
```

- ABCs catch careless usage errors

```
class UnixPipe(IStream):
    def recv(self, maxbytes=None):
        ...
    def write(self, data):
        pass

>>> p = UnixPipe()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class
UnixPipe with abstract methods read
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Handler Classes

- Sometimes code will implement a general purpose algorithm, but will defer certain steps to a separately supplied handler object
- Sometimes known as the "strategy" design pattern.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Handler Classes

- Example :

```
def print_table(records, fields, formatter):
    formatter.headings(fields)
    for r in records:
        rowdata = [getattr(r, fieldname, 'undef')
                   for fieldname in fields]
        formatter.row(rowdata)
```

Calls to handler methods

- Notice how various steps of the algorithm are deferred to a separate handler object

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Handler Classes

- Handlers have their own class definition

```
class TableFormatter:
    def headings(self, headings):
        raise NotImplementedError
    def row(self, rowdata):
        raise NotImplementedError
```

- The handler only contains the methods that need to be implemented/customized
- Important idea : Decoupling of the class that produces the table from the handler methods

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Handler Classes

- Example Use

```
class TextTableFormatter(TableFormatter):
    def headings(self, headers):
        for h in headers:
            print('%10s' % h, end=' ')
        print()
        print(('-' * 10 + ' ') * len(headers))
    def row(self, rowdata):
        for d in rowdata:
            print('%10s' % d, end=' ')
        print()

formatter = TextTableFormatter(handler)
print_table(portfolio, ['name','shares'], formatter)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Commentary

- The use of handler classes is extremely common throughout the Python standard library (might be the most popular OO design pattern used in Python)
- Rationale : This approach provides flexibility
- Handlers are decoupled from implementation
- Allows handler code to be reused in other contexts (other classes can use the same handler objects).

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Classes as a Template

- A class might implement a general-purpose algorithm, but delegate certain steps to a subclass
- Will illustrate with a simple example

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Template Example

- A class that parses a CSV file into a list

```
class CSVParser:
    def parse(self, filename):
        records = []
        with open(filename) as f:
            rows = csv.reader(f)
            self.headers = next(rows)
            for row in rows:
                record = self.make_record(row)
                records.append(record)
        return records

    def make_record(self, row):
        raise RuntimeError('Must implement')
```

Step that must be implemented

- Note: Class is useless by itself

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Template Example

- Using the template (use inheritance)

```
class DictCSVParser(CSVParser):
    def make_record(self, row):
        return dict(zip(self.headers, row))

parser = DictCSVParser()
portfolio = parser.parse('portfolio.csv')
```

- Critical idea : User defines a small class that supplies the one missing piece, but most of the real functionality is in the base class

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Prefer Functions

- The template pattern is often overcomplicated
- Consider a function + callback instead

```
def parse_csv(filename, make_record):
    records = []
    with open(filename) as f:
        rows = csv.reader(f)
        headers = next(rows)
        for row in rows:
            record = make_record(headers, row)
            records.append(record)
    return records

def make_record(headers, row):
    # User implements
    ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 3.7

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Advanced Inheritance

- Recall: Inheritance is a tool for code reuse (customization and extension)

```
class Parent:
    def spam(self):
        ...

class Child(Parent):
    def spam(self):
        print('Different spam')
        super().spam()
```

- Child classes can customize their parents
- Sometimes see use of super() function (shown)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Multiple Inheritance

- Classes can have multiple parents

- The child will inherit features from all parents
- But, it's a lot sneakier than this

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Cooperative Inheritance

- Python uses "cooperative multiple inheritance"
- Big idea: A child class can specifically arrange its parents to cooperate with each other

```
class Child(Parent1, Parent2, Parent3):
    ...
```

- The order of the parents has significance
- Attribute search may jump parent-to-parent

```
class Child(Parent1, Parent2, Parent3):
    ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Cooperative Inheritance

- Example: Consider this arrangement

```
class Parent:
    def spam(self):
        print('Parent')
```

```
class A(Parent):
    def spam(self):
        print('A')
        super().spam()
```

```
class B(Parent):
    def spam(self):
        print('B')
        super().spam()
```

- Now, this:

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

```
>>> c = Child()
>>> c.spam()
A
B
Parent
>>>
```

it's gone sideways!

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Cooperative Inheritance

- There are applications

- You can make collections of classes that are meant to be stacked together to make more interesting things

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

An Odd Code Reuse

```
class Dog:
    def noise(self):
        return 'Woof'

    def chase(self):
        return 'Chasing!'

class LoudDog(Dog):
    def noise(self):
        return super()\
               .noise().upper()
```

```
class Bike:
    def noise(self):
        return 'On Your Left'

    def pedal(self):
        return 'Pedaling!'

class LoudBike(Bike):
    def noise(self):
        return super()\
               .noise().upper()
```

- Completely unrelated objects
- But, there is a code commonality

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Mixin Classes

- A mixin is a class whose purpose is to add extra functionality to <u>other</u> class definitions
- Idea : If a user implements some basic features in their class, a mixin can be used to fill out the class with extra functionality
- Sometimes used as a technique for reducing the amount of code that must be written

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Mixin Example

- A class with a fragment of code

```
class Loud:
    def noise(self):
        return super().noise().upper()
```

- Not usable in isolation
- Mixes with other classes via inheritance

```
class LoudDog(Loud, Dog):
    pass

class LoudBike(Loud, Bike):
    pass
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

How it works

- Example:

```
class LoudDog(Loud, Dog):
    pass

>>> d = LoudDog()
>>> d.noise()
'WOOF'
>>>
```

super()

- super() moves to the next class
- Allows mixins to combine with arbitrary classes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Use of Mixins

- Mixin classes are sometimes used as a way to add optional features to more basic objects
- For example, added thread support, persistence, etc.
- User assembles an object from the different parts that they're going to use

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 3.8

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Section 4

Inside Python Objects

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Overview

- Inner details on how Python objects work
- Object representation
- Attribute binding
- Type checking
- Descriptors
- Attribute special methods

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Dictionaries Revisited

- A dictionary is a collection of named values

```
stock = {
          'name'   : 'GOOG',
          'shares' : 100,
          'price'  : 490.10
        }
```

- 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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Dicts and Objects

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Dicts and Instances

- A dictionary holds instance data (__dict__)

```
>>> s = Stock('GOOG',100,490.10)
>>> s.__dict__
{'name' : 'GOOG','shares' : 100, 'price': 490.10 }
```

- You populate this dict when assigning to self

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Dicts and Instances

- Critical point : Each instance gets its own private dictionary

```
s = Stock('GOOG',100,490.10)
t = Stock('AAPL',50,123.45)
```

```
{
    'name' : 'GOOG',
    'shares' : 100,
    'price' : 490.10
}
```

```
{
    'name' : 'AAPL',
    'shares' : 50,
    'price' : 123.45
}
```

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Dicts and Classes

- A dictionary holds the members of a class

```
class Stock:
    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
```

```
Stock.__dict__ ──────────►  {
                             'cost' : <function>,
                             'sell' : <function>,
                             '__init__' : <function>,
                            }
```

methods

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Instances and Classes

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

```
>>> s = Stock('GOOG', 100, 490.10)
>>> s.__dict__
{'name':'GOOG','shares':100,'price':490.10 }
>>> s.__class__
<class '__main__.Stock'>
>>>
```

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Instances and Classes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Modifying Instances

- Operations that modify an object always update the underlying dictionary

```
>>> s = Stock('GOOG',100,490.10)
>>> s.__dict__
{'name':'GOOG', 'shares':100, 'price':490.10 }
>>> s.shares = 50
>>> s.date = '6/7/2007'
>>> s.__dict__
{ 'name':'GOOG', 'shares':50, 'price':490.10,
  'date':'6/7/2007'}
>>> del s.shares
>>> s.__dict__
{ 'name':'GOOG', 'price':490.10,  'date':'6/7/2007'}
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Reading Attributes

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

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

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 4.1

Time : 10 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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
- This link simply extends the search process used to find attributes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

The MRO

- The inheritance chain is precomputed and stored in an "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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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 there is an added complication in that there may be many possible search paths

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Multiple Inheritance

- Python uses "cooperative multiple inheritance"
- There are some ordering rules:

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

- Inheritance works in two directions (up the hierarchy, across the list of parents)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Multiple Inheritance

- Multiple inheritance hierarchy is flattened

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

- Calculated using the C3 Linearization algorithm
- A constrained merge sort of parent MROs
- An ordering based on "the rules"
- Note: If you must know, there's also a third rule (first listed parent wins if there's a "tie").

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Multiple Inheritance

- Consider classes with a common parent

```
class Base:
    ...

class A(Base):
    ...

class B(Base):
    ...

class C(Base):
    ...
```

- All children of a common parent go first

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

MRO

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Why super()?

- Always use super() when overriding methods

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

- super() delegates to the <u>next class</u> on the MRO

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

super() Explained

- super() is one of the most poorly understood Python features

```
class A(Base):
    def spam(self):
        Base.spam(self)
```

vs.

```
class A(Base):
    def spam(self):
        super().spam()
```

- These two classes are <u>not</u> the same
- super() binds to the <u>next implementation</u> that is defined according to the instance's MRO
- It's <u>not</u> necessarily the immediate parent

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Designing for Inheritance

- Rule 1: Compatible Method Arguments

- Overridden methods must have a compatible signature across the entire hierarchy
- Remember: super() might not go to the immediate parent
- Tip: If there are varying method signatures, use keyword arguments

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Designing for Inheritance

- Rule 2: Method chains must terminate

- You can't use super() forever--some class has to terminate the search chain

```
class Base:
    def spam(self):
        pass
```

- Typically the role of an abstract base class

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Designing for Inheritance

- Rule 3: use super() everywhere

- Direct parent calls might explode heads

```
class A(Base):
    def spam(self):
        Base.spam(self)    # NO!
```

- If multiple inheritance is used, a direct parent call will probably violate the MRO

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 4.2

Time : 25 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Dicts and Classes (Reprise)

- Recall, a dictionary holds class members

```
class Stock:
    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
```

```
Stock.__dict__ ──────►  {
                         'cost' : <function>,
                         'sell' : <function>,
                         '__init__' : <function>,
                        }
```

methods

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Reading Attributes (Reprise)

- Recall that a two-step process is used to locate attributes on objects

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

- This is mostly correct
- Except for the extra hidden magic (not shown)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Attribute Binding

- Access to attributes of classes involves one extra processing step
- Something known as the "descriptor protocol"
- It's so sneaky that most Python programmers don't even know it exists
- Yet, it holds the whole object system together

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Descriptor Protocol

- Whenever an attribute is accessed on a class, the attribute is checked to see if it is an object that looks like a so-called "descriptor"
- A *descriptor* is an object with one or more of the following special methods

```
d.__get__(obj, cls)
d.__set__(obj, value)
d.__delete__(obj)
```

- If a descriptor is detected, one of the above methods gets triggered on access

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Descriptor Demo

- Here is a class that implements a dummy descriptor (with prints for debugging)

```
class Descriptor:
    def __init__(self, name):
        self.name = name
    def __get__(self, instance, cls):
        print('%s:__get__' % self.name)
    def __set__(self, instance, value):
        print('%s:__set__ %s' % (self.name, value))
    def __delete__(self, instance):
        print('%s:__delete__' % self.name)
```

- Basically, a descriptor is just an object with get, set, and delete methods

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Descriptor Demo

- Descriptors are placed in class definitions

```
class Foo:
    a = Descriptor('a')
    b = Descriptor('b')
```

- Now, watch what happens on access:

```
>>> f = Foo()
>>> f.a
a:__get__
>>> f.a = 42
a:__set__ 42
>>> del f.a
a:__delete__
>>> f.b
b:__get__
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Descriptor Demo

- Descriptors are presented with information about the instance, class, and values

```
class Descriptor:
    def __init__(self, name):
        self.name = name
    def __get__(self, instance, cls):
        print('%s:__get__' % self.name)
    def __set__(self, instance, value):
        print('%s:__set__ %s' % (self.name, value))
    def __delete__(self, instance):
        print('%s:__delete__' % self.name)
```

```
f = Foo()
f.a
f.a = 42
del f.a
```

- Confusion: self is the descriptor itself, instance is the object it's operating on.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Descriptor Storage

- Descriptors store and retrieve data

```
class Descriptor:
    def __init__(self, name):
        self.name = name
    def __get__(self, instance, cls=None):
        return instance.__dict__[self.name]
    def __set__(self, instance, value):
        instance.__dict__[self.name] = value
```

- Example:

```
class Foo(object):
    a = Descriptor('a')
    b = Descriptor('b')

f = Foo()
f.a = 23     # Stores value in f.__dict__['a']
```

Direct manipulation of the instance dictionary

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Descriptor Binding

- Descriptors always override __dict__

```
class Foo:
    a = Descriptor('a')
    b = Descriptor('b')
```

- Modify the instance dict and try accessing

```
>>> f = Foo()
>>> f.__dict__['a'] = 42
>>> f.__dict__
{'a': 42}
>>> f.a
a:__get__
>>>
```

notice how the descriptor runs regardless the value in the instance dictionary

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Who Cares?

- Every major feature of classes is implemented using descriptors
  - Instance methods
  - Static methods (@staticmethod)
  - Class methods (@classmethod)
  - Properties (@property)
  - __slots__
- Descriptors provide the glue that connects instances and classes together in the runtime

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Descriptors in Action

- Recall that . and () are separate operations

```
>>> s = Stock('GOOG',100,490.10)
>>> s.cost
<bound method Stock.cost of <__main__.Stock object at
0x37e250>>
>>> s.cost()
49010.0
>>>
```

- Focus on that "bound method" result
- How did that get created? Magic?
- No, a descriptor did that.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Descriptors in Action

- Behind the scenes of method lookup

class attribute lookup

descriptor check and invocation

```
>>> s = Stock('GOOG',100,490.10)

>>> value = Stock.__dict__['cost']
>>> value
<function cost at 0x378770>
>>> hasattr(value,"__get__")
True
>>> result = value.__get__(s,Stock)
>>> result
<bound method Stock.cost of <__main__.Stock object at
0x37e250>>
>>> result()
49010.0
>>>
```

- Functions are descriptors where __get__() creates the bound method object

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Descriptors and Properties

- Consider a class with a property attribute

```
class Stock:
    @property
    def shares(self):
        return self._shares
    @shares.setter
    def shares(self, value):
        self._shares = value
```

- A property is also a descriptor

```
>>> s = Stock()
>>> p = Stock.__dict__['shares']
>>> p
<property object at 0x3759c0>
>>> p.__set__(s, 100)    # Same as s.shares = 100
>>> p.__get__(s, Stock)  # Same as s.shares
100
>>> s.shares
100
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Descriptors and __slots__

- Consider a class with slots

```
class Foo:
    __slots__ = ('x','y','z')
    ...
```

- Internally, an array is allocated

0	1	2
x	y	z

- Each slot name is used to create a descriptor that simply gets or sets values in the appropriate array position (internals are implemented in C and hard to view though)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Descriptor Commentary

- Descriptors are one of Python's most powerful customizations (you own the dot)
- Experts can create their own custom descriptors and use them to change what happens in the low levels of the object system
- Often used in advanced programming frameworks and as an encapsulation tool

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Descriptor Application

- A common use of descriptors is in describing data (e.g., Object Relational Mapping, etc.)

```
class Stock:
    name   = String('name',maxlen=8)
    shares = Integer('shares')
    price  = Real('price')
```

- Provide more precise control than properties.
- Results in less repetitive code

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Descriptor Application

- Example descriptor code:

```
class Integer:
    def __init__(self, name):
        self.name = name
    def __get__(self, instance, cls):
        return instance.__dict__[self.name]
    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise TypeError('Expected an integer')
        instance.__dict__[self.name] = value
```

- Minor note: __get__() can be omitted if the name exactly matches that in the instance dict

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Tricky Bits with __get__

- __get__ can be accessed in two ways

```
class Foo:
    a = Descriptor('a')
```

- Through an instance (bound)

```
f = Foo()
f.a
```

- On the class definition itself (unbound)

```
Foo.a
```

- Example : Instance vs. class methods

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Tricky Bits with __get__

- Recommended __get__ implementation

```
class Descriptor:
    def __get__(self, instance, cls):
        if instance is None:
            # If no instance given, return the descriptor
            # object itself
            return self
        else:
            # Return the instance value
            return instance.__dict__[self.name]
```

- Always check for presence of an instance (instance). If None, return the descriptor itself

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Method Descriptors

- A weaker descriptor that only has __get__

```
class MethodDescriptor:
    def __get__(self, instance, cls):
        print('Getting!')
```

- Only triggered if obj.__dict__ doesn't match

```
class Foo(object):
    a = MethodDescriptor("a")

>>> f = Foo()
>>> f.a
Getting!
>>> f.__dict__['a'] = 42
>>> f.a
42
>>>
```

Notice how the value in the dictionary hides the descriptor

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Descriptor Naming

- Descriptors can define a name setter

```
class Descriptor:
    def __init__(self, name=None):
        self.name = name

    def __get__(self, instance, cls):
        return instance.__dict__[self.name]

    def __set_name__(self, cls, name):
        self.name = name
```

- Gets information about definition context

```
class Spam(object):
    x = Descriptor()  ──────►  x.__set_name__(Spam, 'x')
```

- Python 3.6+ only

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Descriptor Conflicts

- Multiple descriptors per attribute aren't allowed

```
class Point:
    __slots__ = ('x', 'y')
    x = Integer()
    y = Integer()
    ...

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'x' in __slots__ conflicts with class variable
```

- There can only one descriptor in charge
- Coordination is possible (but tricky)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 4.3

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Attribute Access Methods

- Classes can intercept attribute access
- Set of special methods for setting, deleting, and getting attributes

Get: `obj.x` → `obj.__getattribute__('x')`

↓ (if not found)

`obj.__getattr__('x')`

Set: `obj.x = val` → `obj.__setattr__('x',val)`

Delete: `del obj.x` → `obj.__delattr__('x')`

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

__getattribute__()

- __getattribute__(self,name)
- Called every time an attribute is read
- Default behavior looks for descriptors, checks the instance dictionary, checks bases classes (inheritance), etc.
- If it can't find the attribute after all of those steps, it invokes __getattr__(self,name)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

__getattr__() method

- __getattr__(self,name)
- A failsafe method. Called if an attribute can't be found using the standard mechanism
- Default behavior is to raise AttributeError
- Sometimes customized

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

__setattr__() method

- __setattr__(self,name,value)
- Called every time an attribute is set
- Default behavior checks for descriptors, stores values in the instance dictionary, etc.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

__delattr__() method

- __delattr__(self,name)
- Called every time an attribute is deleted
- Default behavior checks for descriptors and deletes from the instance dictionary

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Customizing Access

- A class can redefine the attribute access methods to implement custom processing
- The most common application of this is for creating wrapper objects, proxies, and other similar kinds of objects

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Example : Proxy

- Consider this class

```
class Proxy:
    def __init__(self,obj):
        self._obj = obj
    def __getattr__(self,name):
        print('getattr:', name)
        return getattr(self._obj, name)
```

- It holds an internal reference to an object
- Attribute access is redirected to held object

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Example : Proxy

- Example use:

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area()
50.26548245743669

>>> p = Proxy(c)
>>> p
<__main__.Proxy object at 0x37f130>
>>> p.radius
getattr: radius
4.0
>>> p.area()
getattr: area
50.26548245743669
>>>
```

Notice how attribute access gets captured by __getattr__ and then redirected to the original object

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Example: Delegation

```
class A:
    def foo(self):
        print('A.foo')
    def bar(self):
        print('A.bar')

class B:
    def __init__(self):
        self._a = A()
    def bar(self):
        print('B.bar')
        self._a.bar()
    def __getattr__(self, name):
        return getattr(self._a, name)
```

- Example:

```
>>> b = B()
>>> b.foo()
A.foo
>>> b.bar()
B.bar
A.bar
>>>
```

- Sometimes used as an alternative to inheritance

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Delegation Caution

- __getattr__ doesn't apply to special methods (e.g., __len__, __getitem__, etc.)
- Must delegate manually (if needed)

```
class B:
    def __init__(self):
        self._a = A()

    def __getitem__(self, index):
        return self._a[index]

    def __getattr__(self, name):
        return getattr(self._a, name)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 4.4

Time : 10 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Section 5

Functions

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Overview

- Function design
- Functional Programming
- Error handling and Logging
- Testing

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Functions

- Functions are a basic building block
- Top-level functions in a module
- Methods of a class
- Almost all of your code should live in a function

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Function Design

- Try to make functions "self-contained"

- Only operate on passed arguments
- Produce the same result for same arguments
- Avoid hidden side-effects
- Goals: Simplicity and Predictability

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Function Invocation

- Names and arguments are the "interface"

```
result = some_func(args)
```

- Calling a function should be intuitive and easy
- It's a core component of making an API

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Naming Conventions

- There is a preferred Python "style"
- Functions should use lowercase names and _

```
def read_data(filename):
    ...
```

Yes

```
def readData(filename):
    ...
```

No

- Use a leading _ for internal/private funcs

```
def _internal_func():
    ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Default Arguments

- Sometimes you want default arguments

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

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

```
d = read_data('data.csv')
e = read_data('data.csv', debug=True)
```

- Default arguments must appear last in definition

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Keyword Arguments

- Prefer keywords for passing optional arguments

```
a = read_data('data.csv', debug=True)    # YES!
b = read_data('data.csv', True)          # NO!
```

- Keywords result in better code clarity
- You can force the use of keyword arguments

```
def read_data(filename, *, debug=False):
    ...
```

All arguments after the *
must be given as by keyword

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Default Values

- Don't use mutable values as defaults

```
def func(a, items=[]):
    items.append(a)
    return items
```

- The default value is only created once for the whole program--mutations are "sticky"

```
>>> func(1)
[1]
>>> func(2)
[1, 2]
>>> func(3)
[1, 2, 3]
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Default Values

- Advice: Only use immutable values such as None, True, False, numbers, or strings

```
def func(a, items=None):
    if items is None:
       items = []
    items.append(a)
    return items
```

- This avoids the problem of the default being modified by accident

```
>>> func(1)
[1]
>>> func(2)
[2]
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Optional Values

- Sometimes you need to express optional values
- Most common convention is to use None

```
name = None          # Not assigned
name = 'Guido'       # Assigned
```

- Test against None when you use it

```
if name is not None:
    print('Hello', name)
```

- Caution: Often error-prone if not careful

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Argument Transformation

- Consider

```
def read_data(filename):
    records = []
    with open(filename) as f:
        for line in f:
            ...
            records.append(record)
    return records
```

- Compare with

```
def read_data(lines):
    records = []
    for line in lines:
        ...
        records.append(record)
    return records
```

Discuss

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Argument Transforms

- This version is far more flexible

```
def read_data(lines):
    records = []
    for line in lines:
        ...
        records.append(record)
    return records
```

```
with open('Data.csv') as f:
    data = read_data(f)

with gzip.open('Data.csv.gz', 'rt') as f:
    data = read_data(f)

r = requests.get('http://place/data.csv')
data = read_data(r.iter_lines(decode_unicode='utf-8')))
```

- Question: Should you embrace flexibility?

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Doc Strings

- Functions should have a doc string

```
def add(x, y):
    '''
    Adds x and y together.
    '''
    return x + y
```

- Feeds the help() command and development tools
- Important: there are no type signatures or other details to help people reading your code. If you can help clarify, that's usually good.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Type Hints (PEP 484)

- Optional annotations can indicate types

```
def add(x:int, y:int) -> int:
    '''
    Adds x and y together.
    '''
    return x + y
```

- The type hints do nothing, but may be useful for code checkers, documentation, IDEs, etc.

```
>>> help(add)
Help on function add in module __main__:

add(x:int, y:int) -> int
    Adds x and y
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 5.1

Time : 10 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Function Results

- Have the function cleanly return a result

- Return multiple values with a tuple if needed

```
def divide(x, y):
    quotient = x // y
    remainder = x % y
    return (quotient, remainder)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Returning Optionals

- Sometimes a function returns an "optional" value
- Common convention is to return None
- Example: Pattern Matching (re module)

```
>>> import re
>>> m = re.match('\d+', 'abc')
>>> print(m)
None
>>> m = re.match('\d+', '123')
>>> print(m)
<_sre.SRE_Match object; span=(0, 3), match='123'>
>>>
```

- For errors: raise an exception instead

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Concurrency

- Functions might execute concurrently (threads)

```
def foo():
    ...

def bar():
    ...

from threading import Thread

t1 = Thread(target=foo)
t1.start()

t2 = thread(target=bar)
t2.start()
```

- Shared state, execution in single interpreter

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Futures

- Represents a future result (to be computed)

```
from concurrent.futures import Future

fut = Future()
```

- To store a result

```
fut.set_result(value)
```

- To wait for a result

```
value = fut.result()
```

- Coordination is required

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Future Example

```
def func(x, y, fut):
    time.sleep(20)
    fut.set_result(x+y)

def caller():
    fut = Future()
    threading.Thread(target=func, args=(2, 3, fut).start()
    result = fut.result()
    print('Got:', result)
```

- Reminder: Concurrent execution of both functions
- This underlying pattern is used in many contexts (threads, async, multiprocessing, etc.)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 5.2

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Functional Programming

- Programming style characterized by
  - Functions
  - No sides effects/mutability
  - Higher order functions

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Higher Order Functions

- Essential features...
  - Functions can accept functions as input
  - Functions can return functions as results
- Python supports both

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Functions as Input

- Consider these two functions

```
def sum_squares(nums):
    total = 0
    for n in nums:
        total += n * n
    return total

def sum_cubes(nums):
    total = 0
    for n in nums:
        total += n ** 3
    return total
```

only difference

- They're almost identical (one line differs)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Functions as Input

- Recognizing commonality is part of abstraction

```
def sum_map(func, nums):
    total = 0
    for n in nums:
        total += func(n)
    return total

def square(x):
    return x * x

nums = [1, 2, 3, 4]
r = sum_map(square, nums)
```

- This version allows any function to be passed
- Sometimes referred to as a "callback function"

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Lambda Functions

- One-expression functions can use lambda

```
def sum_map(func, nums):
    total = 0
    for n in nums:
        total += func(n)
    return total

nums = [1, 2, 3, 4]
result = sum_map(lambda x: x*x, nums)
```

- Creates an anonymous function (on the spot)
- Can only contain a single expression
- No control flow, exceptions, etc.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Partial Application

- Lambda often used to alter function args

```
def distance(x, y):
    return abs(x - y)

>>> distance(10, 20)
10
>>> dist_from10 = lambda y: distance(10, y)
>>> dist_from10(3)
7
>>> dist_from10(14)
4
>>>
```

- functtools.partial

```
from functtools import partial
dist_from10 = partial(distance, 10)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Map-Reduce

- You can sometimes decompose the code further

```
def sum_map(func, nums):
    total = 0
    for n in nums:
        total += func(n)
    return total

nums = [1, 2, 3, 4]
r = sum_map(lambda x: x*x, nums)
```

func(n) Mapping

total += Reduction

- There are two basic operations (+, func)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Map-Reduce

```
def map(func, values):
    result = []
    for x in values:
        result.append(func(x))
    return result

def reduce(func, values, initial=0):
    result = initial
    for n in values:
        result = func(n, result)
    return result

def sum(x, y):
    return x + y

def square(x):
    return x * x

nums = [1, 2, 3, 4]
result = reduce(sum, map(square, nums))
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Commentary

- Subdividing problems into small composable parts is a useful software architecture tool

- Functions are the basic building blocks

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 5.3

Time : 10 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Returning Functions

- Consider the following function

```
def add(x, y):
    def do_add():
        print(f'{x} + {y} -> {x+y}')
    return do_add
```

- A function that returns another function?

```
>>> a = add(3,4)
>>> a
<function do_add at 0x6a670>
>>> a()
3 + 4 -> 7
>>>
```

- Notice that it works, but ponder it...

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Nested Scopes

- Observe how the inner function refers to variables defined by the outer function

```
def add(x, y):
    def do_add():
        print(f'{x} + {y} -> {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()
3 + 4 -> 7
```

Where are the x,y values coming from?

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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(f'{x} + {y} -> {x+y}')
    return do_add
```

- Essential feature : A "closure" retains the values of all variables needed for the function to run properly later on

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Closures

- To make it work, references to the outer variables (bound variables) get carried along with the function

```
def add(x, y):
    def do_add():
        print(f'{x} + {y} -> {x+y}')
    return do_add
```

```
>>> a = add(3, 4)
>>> a.__closure__
(<cell at 0x54f30: int object at 0x54fe0>,
 <cell at 0x54fd0: int object at 0x54f60>)
>>> a.__closure__[0].cell_contents
3
>>> a.__closure__[1].cell_contents
4
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Closures

- Closures only capture used variables

```
def add(x, y):
    result = x + y
    def get_result():
        return result
    return get_result
```

```
>>> a = add(3, 4)
>>> a.__closure__
(<cell at 0x10bb52708: int object at 0x10b5d3610>,)
>>> a.__closure__[0].cell_contents
7
>>>
```

- Carefully observe: x and y are not included (not needed in the function body)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Closures and Mutability

- Closure variables are mutable (nonlocal decl)

```
def counter(n):
    def incr():
        nonlocal n
        n += 1
        return n
    return incr

>>> c = counter(10)
>>> c()
11
>>> c()
12
>>>
```

- Can be used to hold mutable internal state, much like an object or class

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using Closures

- Closures are an essential feature of Python
- Common applications:
  - Alternate evaluation (e.g., "delayed evaluation)
  - Callback functions
  - Code creation ("macros")

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 5.4

Time : 10 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Function Error Checking

- As you know, exceptions indicate errors

```
raise RuntimeError("You're dead")
```

- Exceptions can be caught

```
try:
    statements
    ...
except RuntimeError as e:
    # Handle the runtime error
    ...
```

- The mechanics of exception handling is usually straightforward, but proper usage is often a lot trickier than it looks

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Function Error Checking

- As you know, exceptions indicate errors

```
raise RuntimeError("You're dead")
```

- Exceptions can be caught

```
try:
    statements
    ...
except RuntimeError as e:
    # Handle the runtime error
    ...
```

- The mechanics of exception handling is usually straightforward, but proper usage is often a lot trickier than it looks

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

What Exceptions to Handle?

- Functions should only handle exceptions where recovery is possible (and makes sense):

```
def read_csv(filename):
    f = open(filename)
    for row in csv.reader(f):
        try:
            name = row[0]
            shares = int(row[1])
            price = float(row[2])
        except ValueError as e:
            print('Bad row:', row)
            continue
        ...
```

- Let all other exceptions propagate--they usually indicate a more serious problem

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Example

- Don't worry about things like this

```
>>> read_csv('bogus.csv')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "reader.py", line 10, in read_csv
    f = open(filename)
FileNotFoundError: [Errno 2] No such file or directory:
'bogus.csv'
>>>
```

- There is no sensible recovery. The failure is someone else's problem.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Catching All Errors

- Never catch all exceptions unless you report/ record the actual exception that occurred

```
try:
    # Some complicated operation
    ...
except Exception as e:
    print("Sorry, it didn't work.")
    print("Reason:", e)
    ...
```

- Not reporting actual exception information is the fastest way to create undebuggable code

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Ignoring Errors

- No! No! No!

```
try:
    # Some complicated operation
    ...
except Exception:
    pass
```

- Argh!!!! Boom!

```
try:
    # Some complicated operation
    ...
except Exception:
    # !! TODO
    pass
```

Ariane 5

- Catastrophic failures are often a result of exception handling gone terribly wrong.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Reraising Exceptions

- Log/re-raise

```
try:
    # Some complicated operation
    ...
except Exception as e:
    print("Sorry, it didn't work.")
    print("Reason:", e)
    raise
```

- Useful if you want to do something with the exception, but allow it to propagate

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Wrapped Exceptions

- Wrapping an exception in another exception

```
try:
    ...
except Exception as e:
    raise TaskError('It failed') from e
```

- Unwrapping it later

```
try:
   ...
except TaskError as e:
   print("It didn't seem to work.")
   print("Reason:", e.__cause__)
```

- Forms an exception chain

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Managing Resources

- Take care to manage system resources correctly

```
def read_data(filename):
    f = open(filename)
    try:
        ... do whatever ...
    finally:
        f.close()
```

- A more modern version (context manager)

```
def read_data(filename):
    f = open(filename)
    with f:
        ... do whatever ...
```

- Failure to do this might cause leaky file descriptors, deadlock, or other problems

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

What Exceptions to Raise?

- Applications should have their own exceptions

```
class ApplicationError(Exception):
    pass

class SomeOtherError(ApplicationError):
    pass
```

- Issue: How do you distinguish between programming mistakes and exceptions that you meant to raise?
- Reserve Python's built-in exceptions for programming mistakes. Catch, don't raise.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Return Codes

- Don't use return codes (usually)

```
def read_data(filename):
    # Some complicated thing
    ...
    if error:
        return -1   # Oops
    else:
        return result
```

- Return codes are not the "standard" way of signaling errors in Python
- Callers will often forget and program will crash for a different reason later.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Logging

- Use logging for recording diagnostics

```
import logging
log = logging.getLogger(__name__)

def read_data(filename):
    ...
    try:
        name = row[0]
        shares = int(row[1])
        price = float(row[2])
    except ValueError as e:
        log.warning("Bad row: %s", row)
        log.debug("Reason : %s", e)
```

- Usually a better option than print() functions

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 5.5

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Example Code

- Suppose you have this function

```
# simple.py
def add(x, y):
    '''
    Adds x and y.
    '''
    return x + y

>>> add(2,2)
4
>>> add('hello','world')
'helloworld'
>>>
```

- Let's test it

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Assertions/Contracts

- Assertions are runtime checks

```
def add(x, y):
    '''
    Adds x and y
    '''
    assert isinstance(x, int)
    assert isinstance(y, int)
    return x + y
```

- Function will fail on bad input

```
>>> add(2, 3)
5
>>> add('2', '3')
Traceback (most recent call last):
...
AssertionError
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Assertions/Contracts

- Assertions are not meant to check user inputs
- Should validate program invariants (internal conditions that must always hold true)
- Failure indicates a programming error and assign blame (e.g., to the caller)
- Can be disabled (python -O)

```
bash % python3 -O prog.py
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

unittest Module

- Built-in module used for testing
- Used by the standard library
- Commonly used in other applications
- Will briefly illustrate

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using unittest

- First, you create a separate file

```
# testsimple.py
import simple
import unittest
```

- Then you define testing classes

```
class TestAdd(unittest.TestCase):
     ...
```

- They must inherit from unittest.TestCase

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using unittest

- 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..."

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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 is near y
self.assertAlmostEqual(x,y,places)

# Assert that an exception is raised
with self.assertRaises(SomeError):
    statement1
    statement2
    ...
```

- There are others

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Running unittests

- To run tests, add the following code

```
# 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)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

unittest comments

- 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)
- Look at pytest as an alternative

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 5.6

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Section 6

Working with Code

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Overview

- Advanced function usage/definition
- Introspection
- Code generation (eval, exec)
- Callables

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Function Arguments

- Functions operate on passed arguments

```
def func(x,y,z):
    statements
```

arguments

- There are two calling styles

```
a = func(1,2,3)          # Positional arguments
a = func(x=1,y=2,z=3)    # Keyword arguments
```

- You can mix the two styles

```
a = func(1,z=3,y=2)
```

- Positional args always go first. Each argument gets one and only one value

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Variable Arguments

- Function that accepts any number of args

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

- Here, the arguments get passed as a tuple

```
func(1,2,3,4,5)

def func(x, *args):
         1   (2,3,4,5)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Variable Arguments

- Function that accepts any keyword args

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

- Extra keywords get passed in a dict

```
func(2, 3, flag=True,mode="fast",header="debug")

def func(x, y, **kwargs):
    ...

          { 'flag' : True,
            'mode' : 'fast',
            'header' : 'debug' }
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Variable Arguments

- A function that takes <u>any</u> arguments

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

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Passing Tuples and Dicts

- Tuples can expand into function args

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

- Dictionaries can expand to keyword args

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

func(data, **kwargs)
# Same as func(data,color='red',delimiter=',',width=400)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 6.1

Time : 20 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Scoping Rules

- Programs assign values to variables

```
x = value              # Global variable

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

- Python manages variables in two scopes
  - Globals (assigned outside functions)
  - Locals (assignments inside functions)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Statement Execution

- All statements execute within two scopes (even statements not part of a function)
- Global scope is always the module in which a function is defined
- Local scope is either private to a function or the same as the global scope (for statements executed at module level)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Modifying Globals

- If you want to modify a global variable you must declare it as such in the function

```
x = 42

def func():
    global x
    x = 37
```

- global declaration must appear before use
- Only necessary for globals that will be <u>modified</u> (globals are already readable)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

globals() and locals()

- globals() - Give you a dictionary representing the contents of global scope
- locals() - Gives you a dictionary representing the contents of local scope
- Can use these to inspect the environment in which a statement will execute

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

builtins module

- Built-in functions are in a special module
- Consulted last when looking up names

```
>>> abs(-45)
45
>>> import builtins
>>> builtins.abs(-45)
45
>>>
```

- You can modify it (but probably not advised)

```
>>> builtins.pi = 3.1415926
>>> pi
3.1415926
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 6.2

Time : 15 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Function Objects

- When you define a function, the function itself becomes a kind of object that you can manipulate
- Can assign to variables, place in containers, pass around as data, etc.
- Functions can also be inspected

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Documentation Strings

- First line of function may be string

```
def func(a, b):
    'This function does something.'
    ...
```

- The doc string is stored in __doc__

```
>>> func.__doc__
'This function does something.'
>>>
```

- Help tools look at __doc__, but other programs might examine it for various purposes (testing, etc.)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Annotations/Type Hints

- Arguments and return might be annotated

```
def func(a:int, b:int) -> int:
    ...
```

- Stored in __annotations__

```
>>> func.__annotations__
{'a': <class 'int'>, 'b': <class 'int'>,
 'return': <class 'int'>}
>>>
```

- Annotations do nothing--purely informational for other code that might want to look at them.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Function Attributes

- Little known fact : You can attach arbitrary attributes to a function

```
def func(a, b):
    'This function does something.'
    ...

func.threadsafe = False
func.blah = 42
```

- Under the hood, each function has a dictionary (__dict__) that holds these values
- Useful for code that manipulates functions

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Function Inspection

- Almost every aspect of a function can be inspected if you know where to look

```
def func(a, b, c=42):
    'This function does something.'
    ...
>>> func.__name__
'func'
>>> func.__defaults__
(42,)
>>> func.__code__
<code object func at 0x325f50, file "<stdin>", line 1>
>>> func.__code__.co_argcount
3
>>> func.__code__.co_varnames
('a', 'b', 'c')
```

- Use the dir() function to see more

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

inspect Module

- Use the inspect module to get details about functions in a more useful form

```
def func(a, b, c=42):
    ...

>>> import inspect
>>> sig = inspect.signature(func)
>>> print(sig)
(a, b, c=42)
>>> list(sig.parameters)
['a', 'b', 'c']
>>> sig.parameters['c'].default
42
>>>
```

- Many more module features (not shown)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Signature Binding

- Signatures can be bound to *args, **kwargs

```
sig = inspect.signature(some_func)

args = (1, 2)
kwargs = {'c': 10}

bound = sig.bind(*args, **kwargs)
for name, val in bound.arguments.items():
    print(name, '=', val)
```

- Performs all error checking
- Might simplify code using *args, **kwargs

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 6.3

Time : 15 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

eval() and exec()

- eval(code) - Evaluates an expression

```
>>> x = 10
>>> eval('3*x - 2')
28
>>>
```

- exec(code) - Executes arbitrary statements

```
>>> exec('for i in range(5): print(i)')
0
1
2
3
4
>>>
```

- Code executes in current globals()/locals()

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

eval() and exec()

- Caution: Modifications to local scope are lost

```
def func():
    x = 10
    exec('x = 15; print(x)')   # ---> 15
    print(x)                   # ---> 10    ?????
```

- eval(*expr* [, *globals* [, *locals*])
- exec(*code* [, *globals* [, *locals*])

```
def func():
    x = 10
    loc = locals()
    exec('x = 15; print(x)', globals(), loc)  # ---> 15
    x = loc['x']
    print(x)                                  # ---> 15
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

eval/exec Caution

- Use these features with extreme care
- Overuse will likely make people hate you
- Tricky interaction with scoping/variables
- Potential security issue with untrusted inputs

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 6.4

Time : 15 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Callable Objects

- You can define your own objects that emulate Python functions (e.g., "callables")

```
class Callable:
    def __call__(self,*args,**kwargs):
        print('Calling', args, kwargs)
```

- Must implement __call__ special method

```
>>> c = Callable()
>>> c(2,3,color='red')
Calling (2, 3) {'color': 'red'}
>>>
```

- Free to do anything you want in __call__

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Defining Callables

- Callable objects sometimes defined when you need to have more than just a function (e.g., storing extra data, caching, etc.)

```
class Memoize:
    def __init__(self,func):
        self._cache = {}
        self._func = func
    def __call__(self,*args):
        if args in self._cache:
           return self._cache[args]
        r = self._func(*args)
        self._cache[args] = r
        return r
    def clear(self):
        self._cache.clear()
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 6.5

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Section 7

Metaprogramming

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Introduction

- Writing programs where there is a lot of code replication is usually problematic
- Tedious to write
- Hard to maintain
- Painful if you decide that you need to make a change (or fix a bug) in all of that extremely repetitive code

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Metaprogramming

- Metaprogramming pertains to the problem of writing code that manipulates other code
- Common examples:
  - Macros
  - Wrappers
  - Aspects
- Essentially, it's doing things to code

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Python Metaprogramming

- Major features
  - Decorators
  - Class decorators
  - Metaclasses
- We're going to talk about all of them
- They are not as difficult to grasp as you think

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Decorators

- A decorator is a function that creates a wrapper around another function
- The wrapper is a new function that works exactly like the original function (same arguments, same return value) except that some kind of extra processing is carried out
- Let's see a simple example first

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Wrapper Functions

- Here is a simple function:

```
def add(x, y):
    return x + y
```

- Here is an example of a wrapper function

```
def logged_add(x, y):
    print('Calling add')
    return add(x, y)
```

- Example use:

```
>>> add(3, 4)
7
>>> logged_add(3, 4)
Calling add
7
>>>
```

This extra output is created by the wrapper, but the original function is still called to get the result

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Creating Wrappers

- Insight : You can write a function that makes a wrapper around <u>any</u> function

```
def logged(func):
    # Define a wrapper function around func
    def wrapper(*args, **kwargs):
        print('Calling', func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

- Usage:

```
>>> logged_add = logged(add)
>>> logged_add
<function wrapper at 0x378670>
>>> logged_add(3, 4)
Calling add
7
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Wrappers as Replacements

- When you create a wrapper, you often want to replace the original function with it

```
def add(x, y):
    return x + y

# Replace add with a wrapped version
add = logged(add)
```

- Other code continues to use the original function name, but is unaware that a wrapper has been injected (that's the whole point)

```
>>> add(3, 4)
Calling add
7
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Decorator Concept

- When you replace a function with a wrapper, you are usually giving the function extra functionality
- This process is known as "decoration"
- You are "decorating" a function with some extra features

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Decorator Syntax

- The definition of a function and wrapping almost always occur together

```
def add(x, y):
    return x + y
add = logged(add)
```

- However, it looks weird and is error prone
- The @decorator syntax simplifies it

```
@logged
def add(x, y):
    return x + y
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Decorator Syntax

- Whenever you see a decorator, a function is getting wrapped. That's it
- Example : In classes

```
class MyClass:
    @staticmethod
    def bar():
        ...
    @classmethod
    def spam(cls):
        ...
    @property
    def name(self):
        ...
```

→

```
class MyClass:
    def bar():
        ...
    bar = staticmethod(bar)
    def spam(cls):
        ...
    spam = classmethod(spam)
    def name(self):
        ...
    name = property(name)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using Decorators

- Use a decorator anytime you want to define a kind of "macro" involving function definitions
- There are many possible applications
  - Debugging and diagnostics
  - Avoiding code replication
  - Enabling/disabling optional features

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Timing Measurements

- A decorator that reports execution time

```
import time
def timethis(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        r = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end - start)
        return r
    return wrapper
```

- Usage:

```
@timethis
def bigcalculation():
    statements
    statements
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 7.1

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Advanced Decorators

- There are a few tricky additional details
  - Multiple decorators
  - Decorators and metadata
  - Decorators with arguments

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Multiple Decorators

- You can apply as many decorators as you want

```
@foo
@bar
@spam
def add(x, y):
    return x + y
```

- This is the same as this:

```
add = foo(bar(spam(add)))
```

- To keep your sanity, it's probably not a good idea to go overboard with it

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Function Metadata

- When you define a function, there is some extra information stored (name, doc strings, etc.)

```
def add(x, y):
    'Adds x and y'
    return x + y

>>> add.__name__
'add'
>>> add.__doc__
'Adds x and y'
>>> help(add)
Help on function add in module __main__:

add(x, y)
    Adds x and y
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

The Metadata Problem

- Decorators don't preserve metadata

```
@logged
def add(x, y):
    'Adds x and y'    return x + y

>>> add.__name__
'wrapper'
>>> add.__doc__
>>> help(add)
Help on function wrapper in module __main__:

wrapper(*args, **kwargs)
>>>
```

- This is a problem

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Copying Metadata

- Decorators should copy metadata

manual copying of metadata →

```
def logged(func):
    def wrapper(*args, **kwargs):
        print('Calling', func.__name__)
        return func(*args, **kwargs)
    wrapper.__name__ = func.__name__
    wrapper.__doc__ = func.__doc__
    return wrapper
```

- A better solution : use @wraps

```
from functools import wraps

def logged(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Calling', func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

Copies metadata from func to the wrapper →

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Decorators with Args

- Decorators can accept arguments

```
@decorator(x, y, z)
def func():
    ...
```

- It's mind boggling, but here's what happens

```
def func():
    ...

func = decorator(x, y, z)(func)
```

- The decorator function must return a <u>function</u> which is called to make a wrapper

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Decorators with Args

- Example: Logging with a custom message

```
def logmsg(message):
    def logged(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(message.format(name=func.__name__))
            return func(*args, **kwargs)
        return wrapper
    return logged
```

- Example use:

```
@logmsg('You called {name}')
def add(x, y):
    return x + y
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Decorators with Args

- Example: Logging with a custom message

```
def logmsg(message):
    def logged(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(message.format(name=func.__name__))
            return func(*args, **kwargs)
        return wrapper
    return logged
```

Arguments can be used by the code inside

Outer function takes the arguments

- The outer function is like an enclosing environment that gets added to the decorator to accept the extra arguments

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Decorators with Args

- Example: Logging with a custom message

The same decorator code as before

```
def logmsg(message):
    def logged(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(message.format(name=func.__name__))
            return func(*args, **kwargs)
        return wrapper
    return logged
```

- Inner functions are standard decorator code

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Decorators with Args

- Decorators with args are more general
- You can specialize to a no-argument case

```
logged = logmsg('Calling {name}')

@logged
def add(x, y):
    return x + y
```

- This is subtle, but useful for simplifying code

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 7.2

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Class Decorators

- Decorators can be applied to class definitions

```
@decorator
class MyClass:
    def bar(self):
        ...
    def spam(self):
        ...
```

- It's exactly the same as doing this:

```
class MyClass:
    def bar(self):
        ...
    def spam(self):
        ...
MyClass = decorator(MyClass)
```

- Manipulates or wraps a class

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Class Decorators

- Most class decorators inspect or do something special with the class definition
- Typical prototype

```
def decorator(cls):
    # Do something with cls
    ...
    # Return the original class back
    return cls
```

- Observe: The original class is not replaced

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Example

- Recording all attribute lookups

```
def logged_getattr(cls):
    # Get the original implementation
    orig_getattribute = cls.__getattribute__

    # Replacement method
    def __getattribute__(self, name):
        print('Getting:', name)
        return orig_getattribute(self, name)

    # Attach to the class
    cls.__getattribute__ = __getattribute__
    return cls
```

- This is replacing a method of the class

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Example

```
@logged_getattr
class MyClass:
    def foo(self):
        pass
    def bar(self):
        pass

>>> s = MyClass()
>>> s.x = 23
>>> s.x
Getting: x
>>> s.foo()
Getting: foo
>>> s.bar()
Getting: bar
>>>
```

Notice how all lookups now have logging

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Decoration via Inheritance

- Base classes can observe inheritance

```
class Parent:
    @classmethod
    def __init_subclass__(cls, **kwargs):
        # Do something with cls (the subclass)
        ...

class Child1(Parent):
    pass

class Child2(Parent, a=1, b=2):
    pass
```

- Makes it easier to have implicit class decorators

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 7.3

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Disclaimer

- What follows is Python's most advanced bit
- Took years for programmers to even understand what anyone was talking about
- Even now, the details are somewhat hairy
- Also known as the "killer joke"

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Types

- As you hopefully know, all values in Python have an associated type
- Example:

```
>>> x = 42
>>> type(x)
<class 'int'>
>>> s = 'Hello'
>>> type(s)
<class 'str'>
>>> items = [1,2,3]
>>> type(items)
<class 'list'>
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Type Constructor

- The "type" is usually a callable for creating objects of that type
- Example:

```
>>> items = [1,2,3]
>>> type(item)
<class 'list'>
>>> a = list()          # Create a new list object
>>> a
[]
>>> b = tuple(items)    # Convert to a tuple
>>>
```

- Type conversions like this are common

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Types and Classes

- Classes also define new types

```
class Spam:
    pass

>>> s = Spam()
>>> type(s)
<class '__main__.Spam'>
>>>
```

- It is exactly the same as with built-ins
- The class is the type of instances created
- The class is a callable that creates instances

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Types of Classes

- Classes are instances of types
- Observe by getting the type of a class itself

```
>>> class Spam:
...     pass
...
>>> type(Spam)
<class 'type'>
>>> isinstance(Spam, type)
True
>>>
```

Recall: type() tells you the type of an object. Here we're using it on a class itself.

- This requires some thought, but it should make some sense (a class is simply a type)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Creating Types

- Head explosion: types are represented by their own class (type)

```
class type:
    ...

>>> type
<class 'type'>
>>>
```

- This class creates new "type" objects
- In fact, this is the class that processes class definitions when you define your own classes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Classes Deconstructed

- Consider a class:

```
class Spam(object):
    def __init__(self, name):
        self.name = name
    def yow(self):
        print("Yow!", self.name)
```

- What are its components?
  - Name ("Spam")
  - Base classes (object)
  - Functions (__init__, yow)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Creating a Class

- You can create a class manually from pieces

```
# Define some method functions
def __init__(self,name):
    self.name = name
def yow(self):
    print("Yow!", self.name)

# Make a method table
methods = {'__init__': __init__,
           'yow': yow }

# Make a new type (Spam)
Spam = type('Spam', (object,), methods)
```

- These steps mimic the class statement

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Class Definition Process

- What happens during class definition?

```
class Spam(object):
    def __init__(self, name):
        self.name = name
    def yow(self):
        print("Yow!", self.name)
```

- Step 1: Body of class is captured

```
body = '''
   def __init__(self, name):
        self.name = name
   def yow(self):
        print("Yow!", self.name)
'''
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Class Definition Process

- Step 2: A dictionary is created

```
__dict__ = type.__prepare__('Spam', (object,))
```

- Normally, you get a plain Python dictionary

```
>>> type.__prepare__('Spam', (object,))
{}
>>>
```

- Some extra metadata is inserted

```
__dict__['__qualname__'] = 'Spam'
__dict__['__module__'] = 'modulename'
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Class Definition Process

- Step 3: Class body is executed in the dict

```
exec(body, globals(), __dict__)
```

- The statements in the body run like a script
- Afterwards, __dict__ is populated

```
>>> __dict__
{
  '__init__': <function __init__ at 0x4da10>,
  'yow': <function yow at 0x4dd70>},
  '__qualname__': 'Spam',
  '__module__': 'modulename'
}
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Class Definition Process

- Step 4: Class is constructed from its name, base classes, and the dictionary

```
>>> Spam = type('Spam', (object,), __dict__)
>>> Spam
<class '__main__.Spam'>
>>> s = Spam('Guido')
>>> s.yow()
Yow! Guido
>>>
```

- type(name, bases, dict) constructs a class object

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 7.4

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Metaclasses Defined

- A class that creates classes is called a metaclass
- type is an example of a metaclass

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

The Metaclass Hook

- Python provides a hook that allows you to override the class creation steps
- You can use a different metaclass than "type"
- Using this, you can completely customize what happens when a class is created.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Metaclass Selection

- metaclass keyword argument
- Sets the class used to create the class object

```
class Spam(metaclass=type):
   def __init__(self, name):
       self.name = name
   def yow(self):
       print("Yow!", self.name)
```

- By default, it's set to 'type', but you change it to something else (more shortly)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Metaclass Selection

- Compatibility note: Python 2 is different
- Use the __metaclass__ attribute instead

```
class Spam:
    __metaclass__ = type          # Python 2 only
    def __init__(self, name):
        self.name = name
    def yow(self):
        print("Yow!", self.name)
```

- Comment: Very difficult to provide Py2/3 compatibility due to syntax difference

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Metaclass Inheritance

- If no metaclass is set, Python uses the same type as the base class

```
class Spam(object):
   def __init__(self, name):
       self.name = name
   def yow(self):
       print("Yow!")
```

```
>>> object.__class__
<class 'type'>
>>> type(Foo)
<class 'type'>
```

- Note: this is why you rarely see it

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Creating a New Metaclass

- You inherit from type and redefine methods such as __new__, __prepare__, etc.

```
class mytype(type):
    @staticmethod
    def __new__(meta, name, bases, methods):
        print('Creating class : ', name)
        print('Base classes   : ', bases)
        print('Attributes     : ', list(methods))
        return super().__new__(meta, name, bases, methods)
```

- Then you define a new root-object

```
class myobject(metaclass=mytype):
    pass
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using a Metaclass

- To use the new metaclass, define classes so that they inherit from your root object

```
class Spam(myobject):
    def __init__(self, name):
        self.name = name
    def yow(self):
        print("Yow!", self.name)
```

- You should see your metaclass at work

```
Creating class :  Spam
Base classes   :  (<class '__main__.myobject'>,)
Attributes     :  ['yow', '__module__', '__init__']
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 7.5

Time : 10 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Typical Applications

- Metaclasses allow alteration of the class definition process itself
- Setting up the class definition environment
- Changing instance creation
- Coding conventions/rules

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using a Metaclass

- Metaclasses allow class definitions to be monitored and manipulated
- There are 4 main interception points

```
type.__prepare__(name, bases)
                ↓
type.__new__(type, name, bases, dict)
                ↓
type.__init__(cls, name, bases, dict)
```

class definition

```
type.__call__(cls, *args, **kwargs)
```

instance creation

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Example: Duplicate Check

```
class dupedict(dict):
    def __setitem__(self, key, value):
        assert key not in self, '%s duplicated' % key
        super().__setitem__(key, value)

class dupemeta(type):
    @classmethod
    def __prepare__(cls, name, bases):
        return dupedict()
```

- Example:

```
class A(metaclass=dupemeta):
    def bar(self):
        pass
    def bar(self):
        pass
```

→ Fails! Duplicate

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Example: Decoration

```
def decorator(func):
    ...
    # Decorator
    ...

class meta(type):
    @staticmethod
    def __new__(meta, clsname, bases, dict):
        for key, val in dict.items():
            if callable(val):
                dict[key] = decorator(val)
        return super().__new__(meta, clsname, bases, dict)
```

- This class wraps all methods with a decorator

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Example: Instance Creation

```
class meta(type):
    def __call__(cls, *args, **kwargs):
        print('Creating instance of', cls)
        return super().__call__(*args, **kwargs)
```

- Example:

```
>>> class A(metaclass=meta):
        pass

>>> a = A()
Creating instance of <class '__main__.A'>
>>>
```

- Potentially useful for special cases (singletons, caching, etc.)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Commentary

- Metaclasses are not something you should be defining without really good reasons
- Target audience:
  - Framework builders
  - Library developers
- Consider using a class decorator first
- End users should not be messing around with metaclasses in their own code

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 7.6

Time : 10 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Section 8

Iterators, Generators, and Coroutines

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Iteration

- Iteration defined: 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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Iteration: Protocol

- Iteration

```
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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Delegating Iteration

- Sometimes custom containers will delegate

```
class Portfolio:
    def __init__(self):
        self._holdings = []

    def __iter__(self):
        return self._holdings.__iter__()
```

- Making containers iterable usually a good idea

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Generators

- Generators simplify customized iteration

```
def countdown(n):
    print('Counting down from', n)
    while n > 0:
        yield n
        n -= 1

>>> for i in countdown(5):
...     print('T-minus', i)
...
Counting down from 5
T-minus 5
T-minus 4
T-minus 3
T-minus 2
T-minus 1
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Generator Functions

- Behavior is different than normal func
- Calling a generator function creates an 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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Generator Functions

- Function only executes on next()

```
>>> x = countdown(10)
>>> x
<generator object at 0x58490>
>>> next(x)    # invokes 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()

```
>>> next(x)
9
>>> next(x)
8
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Generator Functions

- When the generator returns, iteration stops

```
>>> next(x)
1
>>> next(x)
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.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Reusing Generators

- Generators are one-time use

```
>>> c = countdown(5)
>>> for x in c:
...     print('T-minus', x)
...
T-minus 5
T-minus 4
T-minus 3
T-minus 2
T-minus 1
>>> for x in c:
...     print('T-minus', x)
...
>>>
```

- You can recreate to start again

```
>>> c = countdown(5)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Reusable Generators

- Subtle trick: Make a class with __iter__()

```
class Countdown:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        n = self.n
        while n > 0:
            yield n
            n -= 1
```

- Every use of iteration makes a new generator

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 8.1

Time : 20 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Producers & Consumers

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

producer

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

consumer

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

- yield produces values
- for consume values

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Generator Pipelines

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

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Generator Pipelines

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

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

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Generator Pipelines

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

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

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

- Pipeline setup (in your program)

```
a = producer()

b = processing(a)

c = consumer(b)
```

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 8.2

Time : 15 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Yield as an Expression

- In generators, yield can be used as an *expression*
- For example, on the right side of an assignment

```
def match(pattern):
    print('Looking for %s' % pattern)
    while True:
        line = yield
        if pattern in line:
            print(line)
```

- Question : What is its value?

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Coroutines

- If you use yield like this, you get a "coroutine"
- It defines a function to which you <u>send</u> values

```
>>> g = match('python')
>>> g.send(None)          # Prime it (explained shortly)
Looking for python
>>> g.send('Yeah, but no, but yeah, but no')
>>> g.send('A series of tubes')
>>> g.send('python generators rock!')
python generators rock!
>>>
```

- Sent values are returned by (yield)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Coroutine Execution

- Execution is the same as for a generator
- When you call a coroutine, nothing happens
- Only runs in response to send() method

```
>>> g = match('python')
>>> g.send(None)
Looking for python
>>>
```

Notice that no output was produced

On first operation, coroutine starts running

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Coroutine Priming

- All coroutines must be "primed" by first calling send(None)
- This advances execution to the location of the first yield expression.

```
def match(pattern):
    print('Looking for %s' % pattern)
    while True:
        line = yield
        if pattern in line:
            print(line)
```

send(None) advances the coroutine to the first yield expression

- At this point, it's ready to receive a value

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using a Decorator

- Remembering the first send() is easy to forget
- Solved by wrapping coroutines with a decorator

```
def consumer(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        cr.send(None)
        return cr
    return start

@consumer
def match(pattern):
    ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Processing Pipelines

- Coroutines can also be used to set up pipes

- You just chain coroutines together and <u>push</u> data through the pipe with send() operations

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

An Example

- A source that mimics Unix 'tail -f'

```
import time
def follow(filename, target):
    f = open(filename)
    f.seek(0,2)      # Go to the end of the file
    while True:
         line = f.readline()
         if line != '':
             target.send(line)
         else:
             time.sleep(0.1)    # Sleep briefly=
```

- A consumer that just prints the lines

```
@consumer
def printer():
    while True:
         line = yield
         print(line, end=' ')
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

An Example

- A filter coroutine

```
@consumer
def match(pattern, target):
    while True:
        line = yield            # Receive a line
        if pattern in line:
            target.send(line)    # Send to next stage
```

- Hooking it up

```
follow('access-log',
       match('python',
       printer()))
```

- A picture

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Dataflow

- With coroutines, you can "fan out"

- More possibilities than a simple pipeline

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 8.3

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Generator Control Flow

- Generators have support for forced termination and exception handling
  - .close() method - terminates
  - .throw() method - raise an exception
- Examples follow

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Closing a Generator

- Use .close() method to shutdown

```
g = genfunc()     # A generator
...
g.close()
```

- This raises GeneratorExit at yield

```
def genfunc():
    ...
    try:
        yield item
    except GeneratorExit:
        # .close() was invoked
        # perform cleanup (if any)
        ...
        return
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Raising Exceptions

- Use .throw(type [,val [, tb]]) for exceptions

```
g = genfunc()     # A generator
...
g.throw(RuntimeError, "You're dead")
```

- This raises an exception at the yield

```
def genfunc():
    ...
    try:
        yield item
    except RuntimeError as e:
        # Handle the exception
        ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 8.4

Time : 10 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Managed Generators

- Observation: A generator function can not execute solely by itself. It must be driven by something else (e.g., for-loop, send(), etc)
- Observation: The yield statement represents a point of preemption. Generators suspend at the yield and don't resume until instructed.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Managed Generators

- Idea: A manager will coordinate the execution of a collection of executing generators

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Managed Generators

- Typical applications
  - Concurrency (tasklets, greenlets, etc.)
  - Actors
  - Event simulation
- This is a big topic
- Will give a simple example

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Example : Concurrency

- Define some "task" functions

```
def countdown(n):
    while n > 0:
        print('T-minus', n)
        yield
        n -= 1

def countup(n):
    x = 0
    while x < n:
        print('Up we go', x)
        yield
        x += 1
```

- Carefully observe: just a bare "yield"

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Example : Concurrency

- Instantiate some tasks in a queue

```
tasks = deque([
    countdown(10),
    countdown(5),
    countup(20)
])
```

- Run a little scheduler (the manager)

```
def run():
    while tasks:
        t = tasks.popleft()      # Get a task
        try:
            t.send(None)         # Run to yield
            tasks.append(t)      # Reschedule
        except StopIteration:
            pass
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Example : Concurrency

- Output

```
T-minus 10
T-minus 5
Up we go 0
T-minus 9
T-minus 4
Up we go 1
T-minus 8
T-minus 4
Up we go 2
...
```

- We see tasks cycling, but there are no threads

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 8.5

Time : 20 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Delegating Generation

- Problem: Library functions involving generators

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

```
def countup(end):
    n = 0
    while n < end:
        yield n
        n += 1
```

```
def up_and_down(n):
    countup(n)
    countdown(n)
```

- Problem: It doesn't run...
- Generators can't run on their own.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Delegating Generation

- Option 1: Drive the generator yourself

```
def up_and_down(n):
    for x in countup(n):
        yield x
    for x in countdown(n):
        yield x
```

- You need to manually control each generator with for-loops, send(), throw(), etc.
- Can get quite complicated for coroutines

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Delegating Generation

- Option 2: Let Python drive it (yield from)

```
def up_and_down(n):
    yield from countup(n)
    yield from countdown(n)
```

- Whatever "outer" code runs the generator will take care of it (you don't worry about it)

```
for x in up_and_down(5):   # 0,1,2,3,4,5,4,3,2,1
    ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Async/Await

- Alternate syntax for defining a coroutine (async)

```
async def greeting(name):
    return f'Hello {name}'

>>> g = greeting('Guido')
>>> g
<coroutine object greeting at 0x10b8b8258>
>>> g.send(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: Hello Guido
```

- Syntax for calling a coroutine (await)

```
async def main():
    names = ['Guido', 'Dave', 'Paula']
    for name in names:
        g = await greeting(name)
        print(g)
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Async/Await

- async/await mainly associated with asynchronous I/O, the asyncio module, and related tools
- Provides a better environment for multitasking
- The topic of a whole different course
- See: "Fear and Awaiting in Async"

https://www.youtube.com/watch?v=E-1Y4kSsAFc

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

More Information

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

  http://www.dabeaz.com/generators

- "A Curious Course on Coroutines and Concurrency" tutorial from PyCon'09

  http://www.dabeaz.com/coroutines

- "Generators: The Final Frontier" tutorial from PyCon'14

  http://www.dabeaz.com/finalgenerator

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 8.6

Time : 15 Minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Section 9

Modules and Packages

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Introduction

- You've written some code
- Now you need to organize it
- Possibly give it to others
- How do you do it?

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Modules Revisited

- As you know, every source file is a module

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

- import statement loads and <u>executes</u> a module

```
import foo

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Module Objects

- Modules are objects

```
>>> import foo
>>> foo
<module 'foo' from 'foo.py'>
>>>
```

- A "namespace" for definitions inside

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

- Actually a layer on top of a dictionary (globals)

```
>>> foo.__dict__['grok']
<function grok at 0x1006b6c80>
>>>
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Special Variables

- A few special variables defined in a module

```
__file__        # Name of the source file
__name__        # Name of the module
__doc__         # Module documentation string
```

- Example: "main" check

```
if __name__ == '__main__':
   print('Running as the main program')
else:
   print('Imported as a module using import')
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Import Implementation

- Import in a nutshell (pseudocode)

```
import types

def import_module(name):
    # locate the module and get source code
    filename = find_module(name)
    code = open(filename).read()

    # Create the enclosing module object
    mod = types.ModuleType(name)

    # Run it
    exec(code, mod.__dict__, mod.__dict__)
    return mod
```

- Source is exec'd in module dictionary
- Contents are whatever is left over

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

import statement

- import executes the <u>entire</u> module

```
# bar.py
import foo
```

- It inserts a name reference to the module object into the dictionary <u>used by the code that made the import</u>

```
# bar.py
import foo

foo.grok(2)
```

bar.__dict__

```
{
    ...
    'foo' : <module 'foo'>
    ...
}
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Module Cache

- Each module is loaded only <u>once</u>
- Repeated imports just return a reference to the previously loaded module
- sys.modules is a dict of all loaded modules

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

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Import Caching

- Import (pseudocode)

```
import types
import sys

def import_module(name):
    # Check for cached module
    if name in sys.modules:
        return sys.modules[name]

    filename = find_module(name)
    code = open(filename).read()
    mod = types.ModuleType(name)
    sys.modules[name] = mod

    exec(code, mod.__dict__, mod.__dict__)
    return mod
```

- There is more, but this is basically it

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

from module import

- Selected symbols can be imported locally

```
# bar.py
from foo import grok

grok(2)
```

- Useful for frequently used names
- Confusion: This does <u>not</u> change how import works. The entire module executes and is cached. This merely copies a name.

```
grok = sys.modules['foo'].grok
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

from module import *

- Takes all symbols from a module and places them into the caller's namespace

```
# bar.py
from foo import *

grok(2)
spam('Hello')
...
```

- However, it only applies to names that don't start with an underscore (_)
- _*name* often used when defining non-imported values in a module.

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Module Reloading

- Modules can sometimes be reloaded

```
>>> import foo
...
>>> import importlib
>>> importlib.reload(foo)
<module 'foo' from 'foo.py'>
>>>
```

- It re-executes the module source on top of the already defined module dictionary

```
# pseudocode
def reload(mod):
    code = open(mod.__file__, 'r').read()
    exec(code, mod.__dict__, mod.__dict__)
    return mod
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Module Reloading Danger

- Module reloading is not advised
- Problem: Existing instances of classes will continue to use old code after reload
- Problem: Doesn't update definitions loaded with 'from module import name'
- Problem: Likely breaks code that performs typechecks or uses super()

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Locating Modules

- When looking for modules, Python first looks in the same directory as the source file that's executing the import

- If a module can't be found there, an internal module search path is consulted

```
>>> import sys
>>> sys.path
['',
 '/usr/local/lib/python36.zip',
 '/usr/local/lib/python3.6',
 '/usr/local/lib/python3.6/plat-darwin',
 '/usr/local/lib/python3.6/lib-dynload',
 '/usr/local/lib/python3.6/site-packages']
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

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, Jan 12 2017, 13:20:23)
[GCC 4.2.1 Compatible Apple LLVM 6.1.0 (clang-602.0.53)]
>>> import sys
>>> sys.path
['', '/project/foo/pyfiles',
 '/usr/local/lib/python36.zip', ... ]
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 9.1

10 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Organizing Libraries

- It is standard practice for Python libraries to be organized as a hierarchical set of modules that sit under a top-level package name

```
packagename
packagename.foo
packagename.bar
packagename.utils
packagename.utils.spam
packagename.utils.grok
packagename.parsers
packagename.parsers.xml
packagename.parsers.json
...
```

- Other programming languages have a similar convention (e.g., Java)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Creating a Package

- To create the module library hierarchy, organize files on the filesystem in a directory with the desired structure

```
packagename/
            foo.py
            bar.py
            utils/
                  spam.py
                  grok.py
            parsers/
                  xml.py
                  json.py
...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Creating a Package

- Add __init__.py files to <u>each</u> directory

```
packagename/
            __init__.py
            foo.py
            bar.py
            utils/
                  __init__.py
                  spam.py
                  grok.py
            parsers/
                  __init__.py
                  xml.py
                  json.py
...
```

- These can be empty, but they should exist

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Using a Package

- Once you have the __init__.py files, the import statement should just "work"

```
import packagename.foo
import packagename.parsers.xml

from packagename.parsers import xml
```

- *Almost* everything should work the same way that it did before except that import statements now have multiple levels

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Fixing Relative Imports

- Relative imports of submodules don't work

```
spam/
     __init__.py
     foo.py
     bar.py
```

```
# bar.py
import foo   # Fails (not found)
```

- The issue: Resolving name clashes between top-level packages and submodules

```
spam/
     __init__.py
     os.py
     bar.py
```

```
# bar.py
import os   # ??? (uses stdlib)
```

- imports are always "absolute" (from top level)

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Absolute Imports

- One approach : use absolute imports

```
spam/
     __init__.py
     foo.py
     bar.py
```

- Example :

```
# bar.py

from spam import foo
```

- Notice use of top-level package name

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Package Relative Imports

- Consider a package

```
spam/
     __init__.py
     foo.py
     bar.py
     grok/
         __init__.py
         blah.py
```

- Package relative imports

```
# bar.py

from . import foo          # Imports ./foo.py
from .foo import name      # Load a specific name

from .grok import blah     # Imports ./grok/blah.py
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Package Environment

- Packages define a few useful variables

```
__package__          # Name of the enclosing package
__path__             # Search path for subcomponents
```

- Example:

```
>>> import xml
>>> xml.__package__
'xml'
>>> xml.__path__
['/usr/local/lib/python3.5/xml']
>>>
```

- Useful if code needs to obtain information about its enclosing environment

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 9.2

10 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

__init__.py Usage

- What are you supposed to do in those files?
- Main use: stitching together multiple source files into a "unified" top-level import

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Module Assembly

- Consider two submodules in a package

```
spam/
     foo.py
```

```
# foo.py

class Foo:
    ...
    ...
```

```
     bar.py
```

```
# bar.py

class Bar:
    ...
    ...
```

- Suppose you wanted to combine them

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Module Assembly

- Combine in __init__.py

```
spam/
     foo.py
```

```
# foo.py

class Foo:
    ...
    ...
```

```
     bar.py
```

```
# bar.py

class Bar:
    ...
    ...
```

```
__init__.py
```

```
# __init__.py

from .foo import Foo
from .bar import Bar
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Module Assembly

- Users see a single unified top-level package

```
import spam

f = spam.Foo()
b = spam.Bar()
...
```

- Split across submodules is hidden

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Case Study

- The collections "module"
- It's actually a package with a few components

_collections.so

```
deque
defaultdict
```

_collections_abc.py

```
Container
Hashable
Mapping
...
```

collections/__init__.py

```
from _collections import (
            deque, defaultdict )


from _collections_abc import *

class OrdererDict(dict):
    ...

class Counter(dict):
    ...
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Controlling Exports

- Submodules should define __all__

```
# foo.py

__all__ = ['Foo']

class Foo(object):
    ...
```

```
# bar.py

__all__ = ['Bar']

class Bar(object):
    ...
```

- Controls 'from module import *'
- Allows easy combination in __init__.py

```
# __init__.py
from .foo import *
from .bar import *

__all__ = [ *foo.__all__, *bar.__all__ ]
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Module Splitting

- Suppose you have a large module

```
# spam.py

class Foo:
    ...
    ...


class Bar:
    ...
    ...
```

- You want to split it into multiple files
- But keep it as a single import

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Module Splitting

- Step 1: Turn into a directory with multiple files

```
spam/
     foo.py
```

```
# foo.py

class Foo:
    ...
    ...
```

```
     bar.py
```

```
# bar.py

class Bar:
    ...
    ...
```

- Split the code you wish

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Module Splitting

- Step 2: Stitch back together in __init__.py

```
spam/
     foo.py
```

```
# foo.py

class Foo:
    ...
    ...
```

```
     bar.py
```

```
# bar.py

class Bar:
    ...
    ...
```

```
     __init__.py
```

```
# __init__.py

from .foo import Foo
from .bar import Bar
```

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 9.3

20 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Circular Imports

- Great care must be given to circular imports within a package

```
# spam/base.py

from . import child

class Base:
    ...
```

```
# spam/child.py

from .base import Base

class Child(Base):
    ...
```

← Fails

- Follow the control-flow
- Definition order matters!

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Circular Imports

- You may have to place imports somewhere else

```
# spam/base.py

class Base:
    ...

from . import child
```

```
# spam/child.py

from .base import Base

class Child(Base):
    ...
```

← OK

- Rule of thumb: Avoid circular dependencies
- Keep in mind: Not always practical

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Main Modules

- python -m *module*
- Runs a specified module as a main program

```
spam/
     __init__.py
     foo.py
     bar.py

bash % python3 -m spam.foo      # Runs spam.foo as main
```

- Can use to enclose supporting scripts/applications within a package

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Main Entry Point

- __main__.py designates an entry point
- Makes a package directory executable

```
spam/
     __init__.py
     __main__.py                    # Starting module
     foo.py
     bar.py

bash % python3 -m spam             # Run package as main
```

- More useful than you might think

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Executable Subpackages

- Example

```
spam/
    __init__.py
    foo.py
    bar.py
    test/
        __init__.py
        __main__.py  ──→  bash % python3 -m spam.test
        foo.py
        bar.py
```

- Could have a variety of such tools/utilities embedded within a package
- Nice feature: they stay with the package

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Exercise 9.4

15 minutes

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

More Information

- "Modules and Packages: Live and Let Die" - PyCon 2015 Tutorial

  http://www.dabeaz.com/modulepackage/

- More than you ever wanted to know...

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

Preparing For Distribution

- Suppose you want to give code to others
- Packaging is a constantly evolving topic
- Best bet: Look at the official docs
- https://packaging.python.org/en/latest/

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,

That's it!

- You've reached the end!
- Thanks!
- Is there even more to learn? Yes. Always.
- I welcome your feedback: dave@dabeaz.com
- Courses: https://www.dabeaz.com/courses.html

Copyright (C) 2007-2023 (CC BY-SA 4.0), David Beazley, https://www.dabeaz.com,