Extreme Python

John Strickler

Version 1.0, May 2023

Table of Contents

About Extreme Python

Course Outline

NOTE The actual schedule varies with circumstances. The last day may include *ad hoc* topics requested by students

Student files

You will need to load some student files onto your computer. The files are in a compressed archive. When you extract them onto your computer, they will all be extracted into a directory named **pyextusgs**. See the setup guides for details.

What’s in the files?

pyextusgs contains all files necessary for the class
pyextusgs/EXAMPLES/ contains the examples from the course manuals.
pyextusgs/ANSWERS/ contains sample answers to the labs.
pyextusgs/DATA/ contains data used in examples and answers
pyextusgs/SETUP/ contains any needed setup scripts (may be empty)

The following folders may be present in some classes:

pyextusgs/BIG_DATA/ contains large data files used in examples and answers
pyextusgs/NOTEBOOKS/ Jupyter notebooks for use in class
pyextusgs/TEMP/ initially empty; used by some examples for output files
pyextusgs/LOGS/ initially empty; used by some examples to write log files

NOTE The student files do not contain Python itself. It will need to be installed separately. This may already have been done.

Examples

Most of the examples from the course manual are provided in EXAMPLES subdirectory.

It will look like this:

Example

cmd_line_args.py

```python
import sys   # Import the sys module

print(sys.argv) # Print all parameters, including script itself

name = sys.argv[1]  # Get the first actual parameter
print("name is", name)
```

cmd_line_args.py apple mango 123

```
['/Users/jstrick/curr/courses/python/common/examples/cmd_line_args.py', 'apple', 'mango',
'123']
name is apple
```

Appendices

Classroom etiquette

Remote learning

- Mic off when you’re not speaking. If multiple mics are on, it makes it difficult to hear
- The instructor doesn’t know you need help unless you let them know via voice or chat.
- It’s ok to ask for help a lot.
  - Ask questions. Ask questions. Ask questions.
  - ***INTERACT*** with the instructor and other students.
- Log off the remote S/W at the end of the day

In-person learning

- Noisemakers off
- No phone conversations
- Come and go quietly during class.

Please turn off cell phone ringers and other noisemakers.

If you need to have a phone conversation, please leave the classroom.

We’re all adults here; feel free to leave the clasroom if you need to use the restroom, make a phone call, etc. You don’t have to wait for a lab or break, but please try not to disturb others.

IMPORTANT Please do not bring any exploding penguins to class. They might maim, dismember, or otherwise disturb your fellow students.

Chapter 1: Using Visual Studio Code with Python

Objectives

- Use VS Code to develop Python applications

About Visual Studio Code

Visual Studio Code (AKA "Code" or "VS Code") is a full-featured IDE for programming. It is free, lightweight, and works across common platforms and with many languages.

Visual Studio Code Features

- Autocomplete (AKA IntelliSense^{tm})
- Autoindent
- Syntax checking/highlighting
- Debugging
- git integration
- Code navigation
- Command palette (easily find any command)
- Smart search-and-replace
- Project management
- Split views
- Zen mode (hide UI details)
- Code snippets (macros)
- Variable explorer
- Integrated Python console
- Interpreter configuration
- Unit testing tools
- Keyboard shortcuts
- Many powerful extensions
- Works with many languages
- Free

Getting started

Installing

If VS Code is not already installed, download and install from https:/code.visualstudio.com .

Adding Python extensions

On some platforms, VS Code comes with Python extensions already installed. Otherwise, please follow these steps:

1. Start VS Code
2. Go to **Extensions** in sidebar on left
3. Search for "python"
4. Select and install the Python extension from Microsoft

Configuration

A little configuration makes VS Code more convenient.

Auto save

Select **File › Auto Save** to turn on Auto Save, which saves your file as you type. You can also configure **Auto Save** permanently in user settings.

Useful settings

Many aspects of VS Code can be configured. For these, go to **File › Preferences › Settings** (or **Code › Preferences › Settings** on Mac).

Launch folder

For convenience, we can set VS Code to run scripts from the folder that contains the scripts.

1. Search for "execute in"
2. Find the setting for **Execute in File Dir**
3. Check the box

Activate environment

For class, it is useful to skip activating the Python environment for all Terminal windows.

1. Search for "Activate Environment"
2. Find the setting for **Activate Environment**
3. Uncheck the box

Minimap

If you do not want to use the minimap (the guide strip along the right margin), you can turn it off.

1. Search for "minimap enabled"
2. Uncheck the box

Font size

You can change the theme (text colors, etc) or font size of VS Code

1. Search for "editor font size"
2. Find setting for **Editor: Font Size**
3. Set font size to desired value

Themes

You can also set the overall theme (colors, font, etc)

1. Go to **File** › **Preferences** › **Themes** › **Color Themes**
2. Choose a new theme

NOTE **Visual Studio Code** may be already setup and configured.

Opening a project

To open a folder, choose **File › Open Folder…**. When you open a folder, it automatically becomes a VS Code *workspace*. This means that you can configure settings specific to this folder and any folders under it.

Generally speaking, this is great for managing individual projects.

Creating a Python script

Use the `File` menu

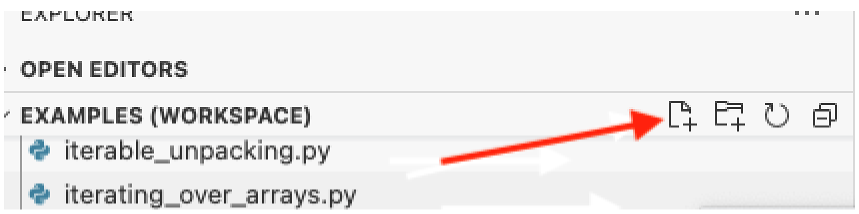
Go to **File › New File…**.

Type a name in the blank, including the `.py` extension. You'll get a file dialog for where to save it.

You can also just select "Python File" and Code will create a new editor window named `untitled-n`, where *n* is a unique number. You can then use **File › Save** to save the file with a permanent name.

Use the "new file" icon

Click on the "new file" icon next to the folder/workplace name in the Explorer panel.



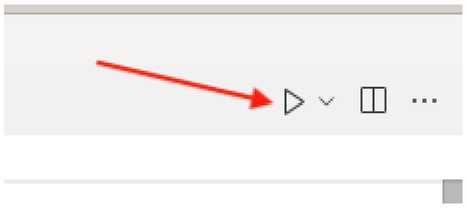
Click on "select a language" and choose Python. As above, it will create a new file named `untitled-n`. Use **File › Save** to save the file.

WARNING This will create a file in the same folder as the currently open file.

Running the script

Click the "run" icon

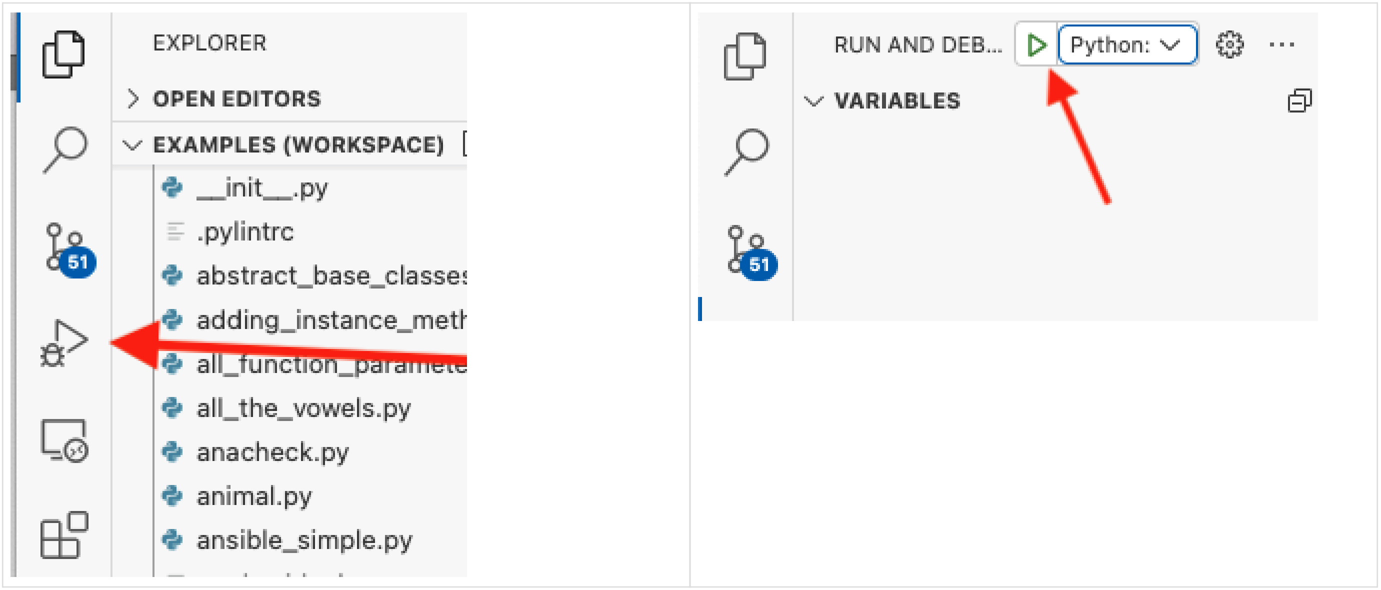
The easiest way to run a script is to click the "Run Python File" icon next to the editor tabs



Once you have run the file, you can re-run it by pressing up-arrow in the terminal window. This makes it easy to add arguments.

Use Run/Debug panel

You can open the Run/Debug panel, and then click on the green arrow to run the script. This uses the default launch configuration, which can be customized.



TIP You can create custom launch configurations to fine-tune how you want to launch each script.

Use Run menu

The Run menu is essentially a shortcut for the tools on the Run/Debug panel. You can choose to run with or without debugging. If you have no breakpoints set, there is not much difference.

Chapter 1 Exercises

Exercise 1-1 (hellocode.py)

In VS Code, create a new script named `hellocode.py`. Put one or more `print()` calls in it and run it with VS Code

Now run it from the command line as well (not in the VS Code integrated terminal).

Chapter 2: Virtual Environments

Objectives

- Learn what virtual environments are
- Understand why you should use them
- Implement virtual environments

Why do we need virtual environments?

Consider the following scenario:

Mary creates an app using Python modules **spamlib** and **hamlib**. She builds a distribution package and gives it to Paul. He installs **spamlib** and **hamlib** on his computer, which has a compatible Python interpreter, and then installs Mary’s app. It starts to run, but then crashes with errors. What happened?

In the meantime, since Mary created her app, the author of **spamlib** removed a function "that almost noone used". When Paul installed **spamlib**, he installed the latest version, which does not have the function. One possible fix is for Paul to revert to an older version of **spamlib**, but suppose he has another app that uses the newer version? This can get very messy very quickly.

The solution to this problem is a **virtual environment**.

What are virtual environments?

A virtual environment starts with a snapshot (copy) of a plain Python installation, before any other libraries are added. It is used to isolate a particular set of modules that will successfully run a given application.

Each application can have its own virtual environment, which ensures that it has the required versions of dependencies. Virtual environments do not have to be in the same folder or folder tree as projects that use them, and multiple projects can use the same virtual environment. However, it’s best practice for each project to have its own.

There are many tools for creating and using virtual environments, but the primary ones are **pip** and the **venv** module.

Preparing the virtual environment

Before creating a virtual environment, it is best to start with a "plain vanilla" Python installation of the desired release. You can use the base Python bundle from https://www.python.org/downloads, or the Anaconda bundle from https://www.anaconda.com/distribution.

You won't use this installation directly — it will be more of a reference installation.

Creating the environment

The **venv** module provides the tools to create a new virtual environment. Basic usage is

```
python -m venv environment_name
```

This creates a virtual environment named *environment_name* in the current directory. It is a copy of the required parts of the original installation.

The virtual environment does not need to in the same location as your application. It is common to create a folder named *.envs* under your home folder to contain all your virtual environments.

Activating the environment

To use a virtual environment, it must be **activated**. This means that it takes precedence over any other installed Python version. This is is implemented by changing the PATH variable in your operating system's environment to point to the virtual copy.

venv will put the name of the environment in the terminal prompt.

Activating on Windows

To **activate** the environment on a Windows system, run the **activate.bat** script in the **Scripts** folder of the environment.

```
environment_name\Scripts\activate
```

Activating on non-Windows

To activate the environment on a Linux, Mac, or other Unix-like system, source the **activate** script in the **bin** folder of the environment. This must be *sourced* — run the script with the **source** builtin command, or the `.` shortcut

```
source environment_name/bin/activate
or
. environment_name/bin/activate
```

Deactivating the environment

When you are finished with an environment, you can deactivate it with the **deactivate** command. It does not need the leading path.

venv will remove the name of the environment from the terminal prompt.

Deactivating on Windows

Run the deactivate.bat script:

```
deactivate
```

Deactivating on non-Windows

Run the deactivate shell script:

```
deactivate
```

TIP To run an app with a particular environment, create a batch file or shell script that activates the environment, then runs the app with the environment’s interpreter. When the script is finished, it should deactivate the environment.

Freezing the environment

When ready to share your app, specify the dependencies by running the **pip freeze** command. This will create a list of all the modules you have added to the virtual environments, and with their current versions. Since the output normally goes to *stdout*, it is conventional to redirect the output to a file named **requirements.txt**.

```
pip freeze > requirements.txt
```

Now you can provide your requirements.txt file to anyone who plans to run your app, and they can create a Python environment with the same versions of all required modules.

Duplicating an environment

When someone sends you an app with a requirements.txt file, it is easy to reproduce their environments. Install Python, then use pip to add the modules from requirements.txt. pip has a `-r` option, which says to read a file for a list of modules to be installed.

```
pip -r requirements.txt
```

That will add the module dependencies with the correct versions.

The pipenv/conda/virtualenv/PyCharm swamp

There are more tools to help with virtual environment management, and having them all available can be confusing. You can always just use pip and venv, as described above.

Here are a few of these tools, and what they do

`conda`

A tool provided by Anaconda that replaces both pip and venv. It assumes you have the Anaconda bundle installed.

`pipenv`

Another tool that replaces both pip and venv. It is very convenient, but has some annoying issues.

`virtualenv`

The original name of the **venv** module.

`PyCharm`

A Python IDE that will create virtual environments for you. It does not come with Python itself.

`virtualenvwrapper`

A workflow manager that makes it more convenient to switch from one environment to another.

Chapter 2 Exercises

Exercise 2-1

Create a virtual environment named **spam** and activate it.

Install the **roman** module.

Create a **requirements.txt** file.

Deactivate **spam**.

Create another virtual environment named **ham** and activate it.

Using **requirements.txt**, make the **ham** environment the same as **spam**.

Chapter 3: Pythonic Programming

Objectives

- Learn what makes code "Pythonic"
- Understand some Python-specific idioms
- Create lambda functions
- Perform advanced slicing operations on sequences
- Distinguish between collections and generators

The Zen of Python

> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.
> Unless explicitly silenced.
> In the face of ambiguity, refuse the temptation to guess.
> There should be one-- and preferably only one --obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.
> Although never is often better than **right** now.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea — let's do more of those!
>
> — Tim Peters, from PEP 20

Tim Peters is a longtime contributor to Python. He wrote the standard sorting routine, known as **timsort**.

The above text is printed out when you execute the code `import this`. Generally speaking, if code follows the guidelines in the Zen of Python, then it's Pythonic.

Tuples

- Fixed-size, read-only
- Collection of related items
- Supports some sequence operations
- Think 'struct' or 'record'

A **tuple** is a collection of related data. While on the surface it seems like just a read-only list, it is used when you need to pass multiple values to or from a function, but the values are not all the same type

To create a tuple, use a comma-separated list of objects. Parentheses are not needed around a tuple unless the tuple is nested in a larger data structure.

A tuple in Python might be represented by a struct or a "record" in other languages.

While both tuples and lists can be used for any data:

- Use a list when you have a collection of similar objects.
- Use a tuple when you have a collection of related objects, which may or may not be similar.

TIP To specify a one-element tuple, use a trailing comma, otherwise it will be interpreted as a single object: color = 'red',

Example

```
hostinfo = ( 'gemini','linux','ubuntu','hardy','Bob Smith' )

birthday = ( 'April',5,1978 )
```

Iterable unpacking

- Copy iterable to list of variables
- Frequently used with list of tuples
- Make code more readable

When you have an iterable such as a tuple or list, you access individual elements by index. However, `spam[0]` and `spam[1]` are not so readable compared to `first_name` and `company`. To copy an iterable to a list of variable names, just assign the iterable to a comma-separated list of names:

```
birthday = ( 'April',5,1978 )
month, day, year = birthday
```

You may be thinking "why not just assign to the variables in the first place?". For a single tuple or list, this would be true. The power of unpacking comes in the following areas:

- Looping over a sequence of tuples
- Passing tuples (or other iterables) into a function

Example

unpacking_people.py

```
people = [
    ('Melinda', 'Gates', 'Gates Foundation', '1964-08-15'),
    ('Steve', 'Jobs', 'Apple', '1955-02-24'),
    ('Larry', 'Wall', 'Perl', '1954-09-27'),
    ('Paul', 'Allen', 'Microsoft', '1953-01-21'),
    ('Larry', 'Ellison', 'Oracle', '1944-08-17'),
    ('Grace', 'Hopper', 'COBOL', '1906-12-09'),
    ('Bill', 'Gates', 'Microsoft', '1955-10-28'),
    ('Mark', 'Zuckerberg', 'Facebook', '1984-05-14'),
    ('Sergey','Brin', 'Google', '1973-08-21'),
    ('Larry', 'Page', 'Google', '1973-03-26'),
    ('Linus', 'Torvalds', 'Linux', '1969-12-28'),
]


# Unpack each tuple into four variables.
for first_name, last_name, product, dob in people:
    print("{} {}".format(first_name, last_name))
```

unpacking_people.py

```
Melinda Gates
Steve Jobs
Larry Wall
Paul Allen
Larry Ellison
Grace Hopper
Bill Gates
Mark Zuckerberg
Sergey Brin
Larry Page
Linus Torvalds
```

Extended iterable unpacking

- Allows for one "wild card"
- Allows common "first, rest" unpacking

When unpacking iterables, sometimes you want to grab parts of the iterable as a group. This is provided by extended iterable unpacking.

One (and only one) variable in the result of unpacking can have a star prepended. This variable will be a list of all values not assigned to other variables.

Example

extended_iterable_unpacking.py

```
values = ['a', 'b', 'c', 'd', 'e']  # values has 6 elements

x, y, *z = values  # {splat} takes all extra elements from iterable
print("x: {}    y: {}    z: {}".format(x, y, z))
print()

x, *y, z = values  # {splat} takes all extra elements from iterable
print("x: {}    y: {}    z: {}".format(x, y, z))
print()

*x, y, z = values  # {splat} takes all extra elements from iterable
print("x: {}    y: {}    z: {}".format(x, y, z))
print()

people = [
    ('Bill', 'Gates', 'Microsoft'),
    ('Steve', 'Jobs', 'Apple'),
    ('Paul', 'Allen', 'Microsoft'),
    ('Larry', 'Ellison', 'Oracle'),
    ('Mark', 'Zuckerberg', 'Facebook'),
    ('Sergey', 'Brin', 'Google'),
    ('Larry', 'Page', 'Google'),
    ('Linux', 'Torvalds', 'Linux'),
]

for *name, _ in people:  # name gets all but the last field
    print(name)
print()
```

extended_iterable_unpacking.py

```
x: a    y: b    z: ['c', 'd', 'e']

x: a    y: ['b', 'c', 'd']    z: e

x: ['a', 'b', 'c']    y: d    z: e

['Bill', 'Gates']
['Steve', 'Jobs']
['Paul', 'Allen']
['Larry', 'Ellison']
['Mark', 'Zuckerberg']
['Sergey', 'Brin']
['Larry', 'Page']
['Linux', 'Torvalds']
```

Unpacking function arguments

- Go from iterable to list of items
- Use * or **

Sometimes you need the other end of iterable unpacking. What do you do if you have a list of three values, and you want to pass them to a method that expects three positional arguments? One approach is to use the individual items by index. A more Pythonic approach is to use * to *unpack* the iterable into individual items:

Use a single asterisk to unpack a list or tuple (or similar iterable); use two asterisks to unpack a dictionary or similar.

In the example, see how the list **HEADINGS** is passed to .format(), which expects individual parameters, not *one parameter* containing multiple values.

Example

unpacking_function_args.py

```
#

people = [ # list of 4-element tuples
    ('Joe', 'Schmoe', 'Burbank', 'CA'),
    ('Mary', 'Brown', 'Madison', 'WI'),
    ('Jose', 'Ramirez', 'Ames', 'IA'),
]

def display_person(first_name, last_name, city, state): # function that takes 4
parameters
    print("{} {} lives in {}, {}".format(first_name, last_name, city, state))

for person in people:  # person is a tuple (one element of people list)
    display_person(*person)  # {splat}person unpacks the tuple into four individual
parameters
```

unpacking_function_args.py

```
Joe Schmoe lives in Burbank, CA
Mary Brown lives in Madison, WI
Jose Ramirez lives in Ames, IA
```

Example

shoe_sizes.py

```
#
BARLEYCORN = 1 / 3.0
CM_TO_INCH = 2.54
MENS_START_SIZE = 12
WOMENS_START_SIZE = 10.5

LINE_FORMAT = '{:6.1f} {:8.2f} {:8.2f}'
HEADER_FORMAT = '{:>6s} {:>8s} {:^8s}'
HEADINGS = ['Size', 'Inches', 'CM']

SIZE_RANGE = []
for i in range(6, 14):
    SIZE_RANGE.extend([i, i + .5])

def main():
    for heading, flag in [("MEN'S", True), ("WOMEN'S", False)]:
        print(heading)
        print((HEADER_FORMAT.format(*HEADINGS)))  # format expects individual arguments
for each placeholder; the asterisk unpacks HEADINGS into individual strings
        for size in SIZE_RANGE:
            lengths = get_length(size, flag)
            print(LINE_FORMAT.format(size, *lengths))

        print()

def get_length(size, mens=True):
    start_size = MENS_START_SIZE if mens else WOMENS_START_SIZE

    inches = start_size - ((start_size - size) * BARLEYCORN)
    cm = inches * CM_TO_INCH
    return inches, cm

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

shoe_sizes.py

```
MEN'S
  Size   Inches    CM
   6.0    10.00    25.40
   6.5    10.17    25.82
   7.0    10.33    26.25
   7.5    10.50    26.67
   8.0    10.67    27.09
   8.5    10.83    27.52
```

...

The sorted() function

- Returns a sorted copy of any collection
- Customize with named keyword parameters

```
key=
reverse=
```

The sorted() builtin function returns a sorted copy of its argument, which can be any iterable.

You can customize sorted with the **key** parameter.

Example

basic_sorting.py

```
"""Basic sorting example"""

fruits = ["pomegranate", "cherry", "apricot", "date", "Apple", "lemon", "Kiwi",
          "ORANGE", "lime", "Watermelon", "guava", "papaya", "FIG", "pear", "banana",
          "Tamarind", "persimmon", "elderberry", "peach", "BLUEberry", "lychee",
          "grape"]

sorted_fruit = sorted(fruits)  # sorted() returns a list

print(sorted_fruit)
```

basic_sorting.py

```
['Apple', 'BLUEberry', 'FIG', 'Kiwi', 'ORANGE', 'Tamarind', 'Watermelon', 'apricot',
'banana', 'cherry', 'date', 'elderberry', 'grape', 'guava', 'lemon', 'lime', 'lychee',
'papaya', 'peach', 'pear', 'persimmon', 'pomegranate']
```

Custom sort keys

- Use **key** parameter
- Specify name of function to use
- Key function takes exactly one parameter
- Useful for case-insensitive sorting, sorting by external data, etc.

You can specify a function with the **key** parameter of the sorted() function. This function will be used once for each element of the list being sorted, to provide the comparison value. Thus, you can sort a list of strings case-insensitively, or sort a list of zip codes by the number of Starbucks within the zip code.

The function must take exactly one parameter (which is one element of the sequence being sorted) and return either a single value or a tuple of values. The returned values will be compared in order.

You can use any builtin Python function or method that meets these requirements, or you can write your own function.

TIP The lower() method can be called directly from the builtin object str. It takes one string argument and returns a lower case copy.

```
sorted_strings = sorted(unsorted_strings, key=str.lower)
```

Example

custom_sort_keys.py

```
fruit = ["pomegranate", "cherry", "apricot", "date", "Apple", "lemon",
         "Kiwi", "ORANGE", "lime", "Watermelon", "guava", "papaya", "FIG",
         "pear", "banana", "Tamarind", "persimmon", "elderberry", "peach",
         "BLUEberry", "lychee", "grape"]

def ignore_case(item):  # Parameter is _one_ element of iterable to be sorted
    return item.lower()  # Return value to sort on

fs1 = sorted(fruit, key=ignore_case)  # Specify function with named parameter key
print("Ignoring case:")
print(" ".join(fs1), end="\n\n")

def by_length_then_name(item):
    return (len(item), item.lower())  # Key functions can return tuple of values to
compare, in order

fs2 = sorted(fruit, key=by_length_then_name)
print("By length, then name:")
print(" ".join(fs2))
print()

nums = [800, 80, 1000, 32, 255, 400, 5, 5000]

n1 = sorted(nums)  # Numbers sort numerically by default
print("Numbers sorted numerically:")
for n in n1:
    print(n, end=' ')
print("\n")

n2 = sorted(nums, key=str)  # Sort numbers as strings
print("Numbers sorted as strings:")
for n in n2:
    print(n, end=' ')
print()
```

custom_sort_keys.py

```
Ignoring case:
Apple apricot banana BLUEberry cherry date elderberry FIG grape guava Kiwi lemon lime
lychee ORANGE papaya peach pear persimmon pomegranate Tamarind Watermelon

By length, then name:
FIG date Kiwi lime pear Apple grape guava lemon peach banana cherry lychee ORANGE papaya
apricot Tamarind BLUEberry persimmon elderberry Watermelon pomegranate

Numbers sorted numerically:
5 32 80 255 400 800 1000 5000

Numbers sorted as strings:
1000 255 32 400 5 5000 80 800
```

Example

sort_holmes.py

```python
"""Sort titles, ignoring leading articles"""
books = [
    "A Study in Scarlet",
    "The Sign of the Four",
    "The Hound of the Baskervilles",
    "The Valley of Fear",
    "The Adventures of Sherlock Holmes",
    "The Memoirs of Sherlock Holmes",
    "The Return of Sherlock Holmes",
    "His Last Bow",
    "The Case-Book of Sherlock Holmes",
]


def strip_article(title):  # create function which takes element to compare and returns
comparison key
    title = title.lower()
    for article in 'a ', 'an ', 'the ':
        if title.startswith(article):
            title = title[len(article):]  # remove article by using a slice that starts
after article + space`
            break
    return title


for book in sorted(books, key=strip_article):  # sort using custom function
    print(book)
```

sort_holmes.py

```
The Adventures of Sherlock Holmes
The Case-Book of Sherlock Holmes
His Last Bow
The Hound of the Baskervilles
The Memoirs of Sherlock Holmes
The Return of Sherlock Holmes
The Sign of the Four
A Study in Scarlet
The Valley of Fear
```

Lambda functions

- Short cut function definition
- Useful for functions only used in one place
- Frequently passed as parameter to other functions
- Function body is an expression; it cannot contain other code

A **lambda function** is a brief function definition that makes it easy to create a function on the fly. This can be useful for passing functions into other functions, to be called later. Functions passed in this way are referred to as "callbacks". Normal functions can be callbacks as well. The advantage of a lambda function is solely the programmer’s convenience. There is no speed or other advantage.

One important use of lambda functions is for providing sort keys; another is to provide event handlers in GUI programming.

The basic syntax for creating a lambda function is

```
lambda parameter-list: expression
```

where parameter-list is a list of function parameters, and expression is an expression involving the parameters. The expression is the return value of the function.

A lambda function could also be defined in the normal manner

```
    def function-name(param-list):
        return expr
```

But it is not possible to use the normal syntax as a function parameter, or as an element in a list.

Example

lambda_example.py

```
fruits = ['watermelon', 'lime', 'Apple', 'Mango', 'KIWI', 'apricot', 'LEMON', 'guava']

sorted_fruits = sorted(fruits, key=lambda e: (len(e), e.lower()))  # The lambda function
takes one fruit name and returns a tuple containing the length of the name and the name
in lower case. This sorts first by length, then by name.

print(sorted_fruits)
```

lambda_example.py

```
['KIWI', 'lime', 'Apple', 'guava', 'LEMON', 'Mango', 'apricot', 'watermelon']
```

List comprehensions

- Filters or modifies elements
- Creates new list
- Shortcut for a for loop

A list comprehension is a Python idiom that creates a shortcut for a for loop. It returns a copy of a list with every element transformed via an expression. Functional programmers refer to this as a mapping function.

A loop like this:

```
results = []
for var in sequence:
    results.append(expr)   # where expr involves var
```

can be rewritten as

```
results = [ expr for var in sequence ]
```

A conditional if may be added to filter values:

```
results = [ expr for var in sequence if expr ]
```

Example

listcomp.py

```python
fruits = ['watermelon', 'apple', 'mango', 'kiwi', 'apricot', 'lemon', 'guava']

values = [2, 42, 18, 92, "boom", ['a', 'b', 'c']]

ufruits = [fruit.upper() for fruit in fruits]  # Copy each fruit to upper case

afruits = [fruit for fruit in fruits if fruit.startswith('a')]  # Select each fruit if it
starts with 'a'

doubles = [v * 2 for v in values]  # Copy each number, doubling it

print("ufruits:", " ".join(ufruits))
print("afruits:", " ".join(afruits))
print("doubles:", end=' ')
for d in doubles:
    print(d, end=' ')
print()
```

listcomp.py

```
ufruits: WATERMELON APPLE MANGO KIWI APRICOT LEMON GUAVA
afruits: apple apricot
doubles: 4 84 36 184 boomboom ['a', 'b', 'c', 'a', 'b', 'c']
```

Dictionary comprehensions

- Expression is key/value pair
- Transform iterable to dictionary

A dictionary comprehension has syntax similar to a list comprehension. The expression is a key:value pair, and is added to the resulting dictionary. If a key is used more than once, it overrides any previous keys. This can be handy for building a dictionary from a sequence of values.

Example

dict_comprehension.py

```
animals = ['OWL', 'Badger', 'bushbaby', 'Tiger', 'Wombat', 'GORILLA', 'AARDVARK']

# {KEY: VALUE for VAR ... in ITERABLE if CONDITION}
d = {a.lower(): len(a) for a in animals}  # Create a dictionary with key/value pairs
derived from an iterable

print(d, '\n')

words = ['unicorn', 'stigmata', 'barley', 'bookkeeper']

d = {w:{c:w.count(c) for c in sorted(w)} for w in words} # Use a nested dictionary
comprehension to create a dictionary mapping words to dictionaries which map letters to
their counts (could be useful for anagrams)

for word, word_signature in d.items():
    print(word, word_signature)
```

dict_comprehension.py

```
{'owl': 3, 'badger': 6, 'bushbaby': 8, 'tiger': 5, 'wombat': 6, 'gorilla': 7, 'aardvark':
8}

unicorn {'c': 1, 'i': 1, 'n': 2, 'o': 1, 'r': 1, 'u': 1}
stigmata {'a': 2, 'g': 1, 'i': 1, 'm': 1, 's': 1, 't': 2}
barley {'a': 1, 'b': 1, 'e': 1, 'l': 1, 'r': 1, 'y': 1}
bookkeeper {'b': 1, 'e': 3, 'k': 2, 'o': 2, 'p': 1, 'r': 1}
```

Set comprehensions

- Expression is added to set
- Transform iterable to set — with modifications

A set comprehension is useful for turning any sequence into a set. Items can be modified or skipped as the set is built.

If you don't need to modify the items, it's probably easier to just past the sequence to the **set()** constructor.

Example

set_comprehension.py

```
import re

FILE_PATH = "../DATA/mary.txt"

# NOTE: r'\W+' is a regular expression that splits on anything that isn't a letter,
number, or underscore

with open(FILE_PATH) as mary_in:
    file_contents = mary_in.read()
    s = {w.lower() for w in re.split(r'\W+', file_contents) if w} # Get unique words from
file. Only one line is in memory at a time. Skip "empty" words.
print(s)
```

set_comprehension.py

```
{'sure', 'the', 'mary', 'little', 'was', 'white', 'a', 'had', 'and', 'its', 'to', 'go',
'as', 'everywhere', 'went', 'fleece', 'lamb', 'that', 'snow'}
```

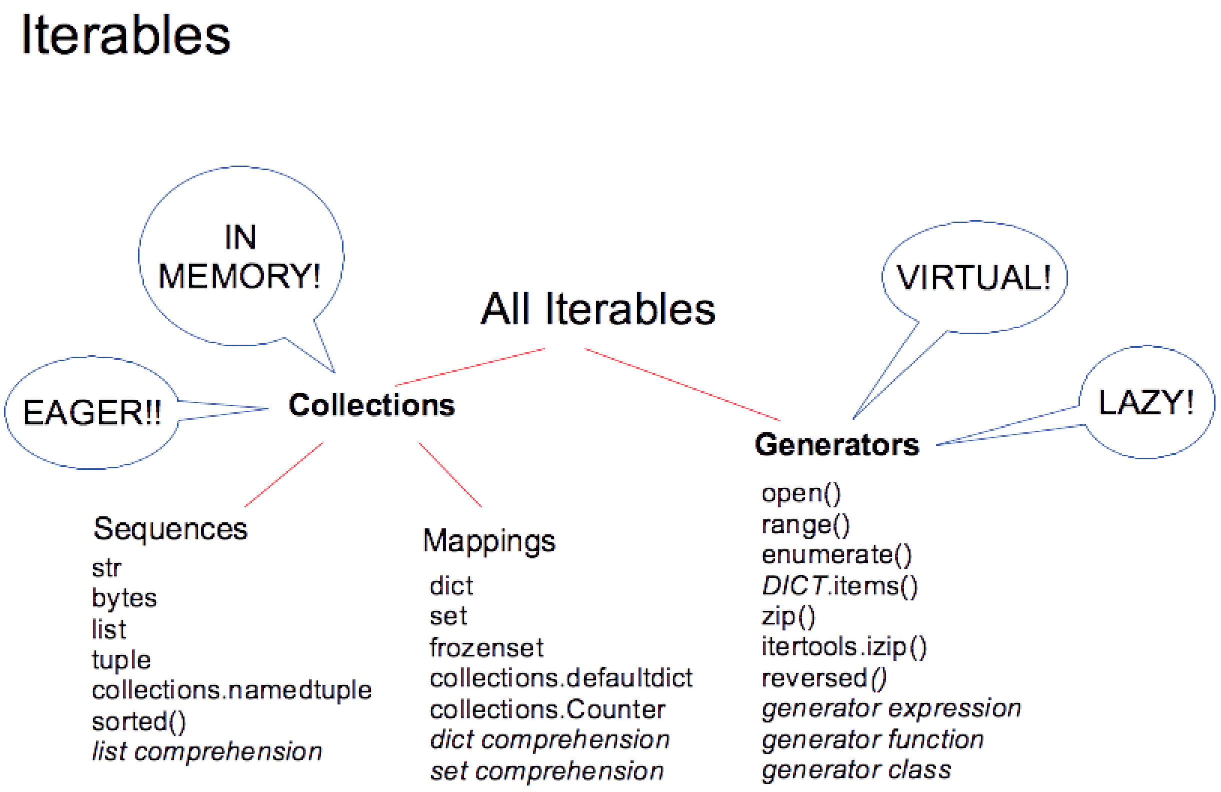
Iterables

- Expression that can be looped over
- Can be collections *e.g.* list, tuple, str, bytes
- Can be generators *e.g.* range(), file objects, enumerate(), zip(), reversed()

Python has many builtin iterables – a file object, for instance, which allows iterating through the lines in a file.

All builtin collections (list, tuple, str, bytes) are iterables. They keep all their values in memory. Many other builtin iterables are *generators*.

A generator does not keep all its values in memory – it creates them one at a time as needed, and feeds them to the for-in loop. This is a Good Thing, because it saves memory.



Generator Expressions

- Like list comprehensions, but create a generator object
- More efficient
- Use parentheses rather than brackets

A generator expression is similar to a list comprehension, but it provides a generator instead of a list. That is, while a list comprehension returns a complete list, the generator expression returns one item at a time.

The main difference in syntax is that the generator expression uses parentheses rather than brackets.

Generator expressions are especially useful with functions like sum(), min(), and max() that reduce an iterable input to a single value:

NOTE There is an implied **yield** statement at the beginning of the expression.

Example

gen_ex.py

```
# sum the squares of a list of numbers
# using list comprehension, entire list is stored in memory
s1 = sum([x * x for x in range(10)])  # using list comprehension, entire list is stored
in memory

# only one square is in memory at a time with generator expression
s2 = sum(x * x for x in range(10))  # with generator expression, only one square is in
memory at a time
print(s1, s2)
print()

with open("../DATA/mary.txt") as page:
    m = max(len(line) for line in page)  # only one line in memory at a time. max()
iterates over generated values
print(m)
```

gen_ex.py

```
285 285

30
```

Generator functions

- Mostly like a normal function
- Use yield rather than return
- Maintains state

A generator is like a normal function, but instead of a return statement, it has a yield statement. Each time the yield statement is reached, it provides the next value in the sequence. When there are no more values, the function calls return, and the loop stops. A generator function maintains state between calls, unlike a normal function.

Example

sieve_generator.py

```python
def next_prime(limit):
    flags = set()  # initialize empty set (to be used for "is-prime" flags

    for i in range(2, limit):
        if i in flags:
            continue
        for j in range(2 * i, limit + 1, i):
            flags.add(j)  # add non-prime elements to set
        yield i  # execution stops here until next value is requested by for-in loop


np = next_prime(200)  # next_prime() returns a generator object
for prime in np:  # iterate over yielded primes
    print(prime, end=' ')
```

sieve_generator.py

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109
113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199
```

Example

line_trimmer.py

```python
def trimmed(file_name):
    with open(file_name) as file_in:
        for line in file_in:
            yield line.rstrip('\n\r')  # 'yield' causes this function to return a
generator object

mary_in = trimmed('../DATA/mary.txt')
for trimmed_line in mary_in:
    print(trimmed_line)
```

line_trimmer.py

```
Mary had a little lamb,
Its fleece was white as snow,
And everywhere that Mary went
The lamb was sure to go
```

String formatting

- Numbered placeholders
- Add width, padding
- Access elements of sequences and dictionaries
- Access object attributes

The traditional (i.e., old) way to format strings in Python was with the % operator and a format string containing fields designated with percent signs. The new, improved method of string formatting uses the format() method. It takes a format string and one or more arguments. The format strings contains placeholders which consist of curly braces, which may contain formatting details. This new method has much more flexibility.

By default, the placeholders are numbered from left to right, starting at 0. This corresponds to the order of arguments to format().

Formatting information can be added, preceded by a colon.

```
{:d}     format the argument as an integer
{:03d}   format as an integer, 3 columns wide, zero padded
{:>25s}  same, but right-justified
{:.3f}   format as a float, with 3 decimal places
```

Placeholders can be manually numbered. This is handy when you want to use a format() parameter more than once.

```
"Try one of these: {0}.jpg {0}.png {0}.bmp {0}.pdf".format('penguin')
```

Example

stringformat_ex.py

```
from datetime import date

color = 'blue'
animal = 'iguana'

print('{} {}'.format(color, animal))  # {} placeholders are autonumbered, starting at 0;
this

fahr = 98.6839832
print('{:.1f}'.format(fahr))  # Formatting directives start with ':'; .1f means format

value = 12345
print('{0:d} {0:04x} {0:08o} {0:016b}'.format(value))  # {} placeholders can be manually
numbered to reuse parameters

data = {'A': 38, 'B': 127, 'C': 9}

for letter, number in sorted(data.items()):
    print("{} {:4d}".format(letter, number))  # :4d means format decimal integer in a
field 4 characters wide
```

stringformat_ex.py

```
blue iguana
98.7
12345 3039 00030071 0011000000111001
A   38
B  127
C    9
```

f-strings

- Shorter syntax for string formatting
- Only available Python 3.6 and later
- Put **f** in front of string

A great new feature, f-strings, was added to Python 3.6. These are strings that contain placeholders, as used with normal string formatting, but the expression to be formatted is also placed in the placeholder. This makes formatting strings more readable, with less typing. As with formatted strings, any expression can be formatted.

Other than putting the value to be formatted directly in the placeholder, the formatting directives are the same as normal Python 3 string formatting.

In normal 3.x formatting:

```
x = 24
y = 32.2345
name = 'Bill Gates'
company = 'Bill Gates'
print("{} founded {}.format(name, company)"
print("{:10s} {:.2f}".format(x, y)
```

f-strings let you do this:

```
x = 24
y = 32.2345
name = 'Bill Gates'
company = 'Bill Gates'
print(f"{name} founded {company})"
print(f"{x:10s} {y:.2f})"
```

Example

f_strings.py

```
city = 'Orlando'
temp = 85
count = 5
avg = 3.4563892382


print(f"It is {temp}\u00B0 in {city}")  # variables inserted into string

# .2f means round a float to 2 decimal points
print(f"count is {count:03d} avg is {avg:.2f}")

print(f"2 + 2 is {2 + 2}")  # any expression is OK
```

f_strings.py

```
It is 85° in Orlando
count is 005 avg is 3.46
2 + 2 is 4
```

Chapter 3 Exercises

Exercise 3-1 (pres_upper.py)

Read the file **presidents.txt**, creating a list of of the presidents' last names. Then, use a list comprehension to make a copy of the list of names in upper case. Finally, loop through the list returned by the list comprehension and print out the names one per line.

Exercise 3-2 (pres_by_dob.py)

Print out all the presidents first and last names, date of birth, and their political affiliations, sorted by date of birth.

Read the **presidents.txt** file, putting the four fields into a list of tuples.

Loop through the list, sorting by date of birth, and printing the information for each president. Use **sorted()** and a lambda function.

Exercise 3-3 (pres_gen.py)

Write a generator function to provide a sequence of the names of presidents (in "FIRSTNAME MIDDLENAME LASTNAME" format) from the presidents.txt file. They should be provided in the same order they are in the file. You should not read the entire file into memory, but one-at-a-time from the file.

Then iterate over the the generator returned by your function and print the names.

Chapter 4: Developer Tools

Objectives

- Run pylint to check source code
- Debug scripts
- Find speed bottlenecks in code
- Compare algorithms to see which is faster

Program development

- More than just coding
  - Design first
  - Consistent style
  - Comments
  - Debugging
  - Testing
  - Documentation

Comments

- Keep comments up-to-date
- Use complete sentences
- Block comments describe a section of code
- Inline comments describe a line
- Don’t state the obvious

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

Use inline comments sparingly. Inline comments should be separated by at least two spaces from the statement; they should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don’t do this:

```
x = x + 1       # Increment x
```

Only use an inline comment if the reason for the statement is not obvious:

```
x = x + 1       # Add one so range() does the right thing
```

The above was adapted from PEP 8

pylint

- Checks many aspects of code
- Finds mistakes
- Rates your code for standards compliance
- Don’t worry if your code has a lower rating!
- Can be highly customized

pylint is a Python source code analyzer which looks for programming errors, helps enforcing a coding standard and sniffs for some code smells (as defined in Martin Fowler’s Refactoring book)

from the pylint documentation

pylint can be very helpful in identifying errors and pointing out where your code does not follow standard coding conventions. It was developed by Python coders at Logilab http://www.logilab.fr.

It has very verbose output, which can be modified via command line options.

pylint can be customized to reflect local coding conventions.

pylint usage:

```
pylint filename(s) or directory
pylint -ry filename(s) or directory
```

The **-ry** option says to generate a full detailed report.

Most Python IDEs have pylint, or the equivalent, built in.

Other tools for analyzing Python code: * pyflakes * pychecker

Customizing pylint

- Use pylint --generate-rcfile
- Redirect to file
- Edit as needed
- Knowledge of regular expressions useful
- Name file \~/.pylintrc on Linux/Unix/OS X
- Use –rcfile file to specify custom file on Windows

To customize pylint, run pylint with only the -generate-rcfile option. This will output a well-commented configuration file to STDOUT, so redirect it to a file.

Edit the file as needed. The comments describe what each part does. You can change the allowed names of variables, functions, classes, and pretty much everything else. You can even change the rating algorithm.

Windows

Put the file in a convenient location (name it something like pylintrc). Invoke pylint with the –rcfile option to specify the location of the file.

pylint will also find a file named pylintrc in the current directory, without needing the -rcfile option.

Non-Windows systems

On Unix-like systems (Unix, Mac OS, Linux, etc.), /etc/pylintrc and ~/.pylintrc will be automatically loaded, in that order.

See docs.pylint.org for more details.

Using pyreverse

- Source analyzer
- Reverse engineers Python code
- Part of pylint
- Generates UML diagrams

pyreverse is a Python source code analyzer. It reads a script, and the modules it depends on, and generates UML diagrams. It is installed as part of the pylint package.

There are many options to control what it analyzes and what kind of output it produces.

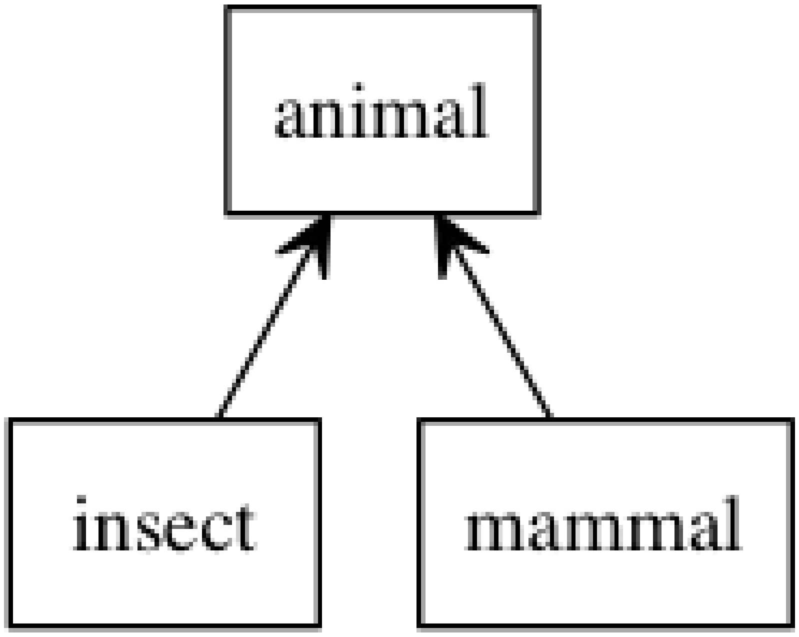
Use `-A' to search all ancestors, `-p` to specify the project name, `-o` to specify output type (e.g., **pdf**, **png**, **jpg**).

NOTE **pyreverse** requires **Graphviz**, a graphics tool that must be installed separately from Python

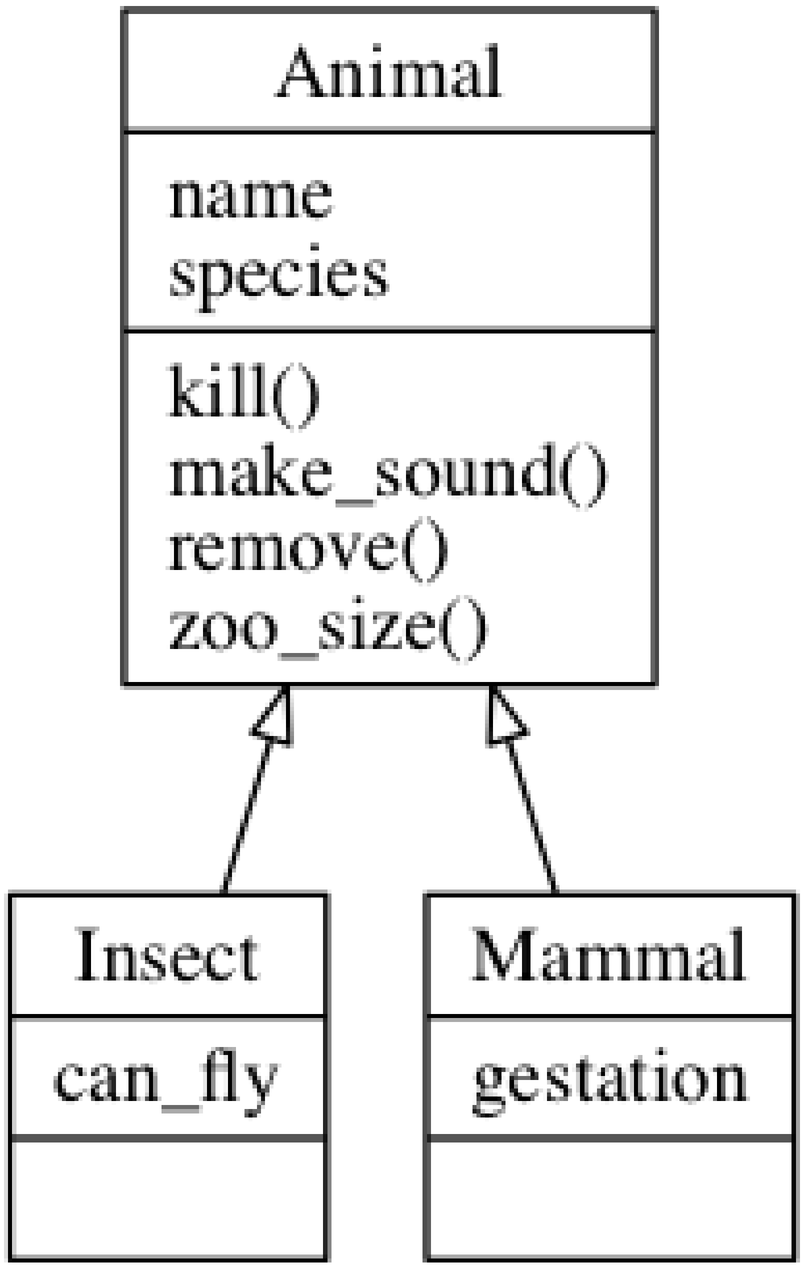
Example

```
pyreverse -o png -p MyProject -A animal.py mammal.py insect.py
```

packages_MyProject.png



classes_MyProject.png



The Python debugger

- Implemented via pdb module
- Supports breakpoints and single stepping
- Based on gdb

While most IDEs have an integrated debugger, it is good to know how to debug from the command line. The pdb module provides debugging facilities for Python.

The usual way to use pdb is from the command line:

```
python -mpdb script_to_be_debugged.py
```

Once the program starts, it will pause at the first executable line of code and provide a prompt, similar to the interactive Python prompt. There is a large set of debugging commands you can enter at the prompt to step through your program, set breakpoints, and display the values of variables.

Since you are in the Python interpreter as well, you can enter any valid Python expression.

You can also start debugging mode from within a program.

Starting debug mode

- Syntax

```
python -m pdb script
```

or

```
import pdb
pdb.run('function')
```

pdb is usually invoked as a script to debug other scripts. For example:

```
python -m pdb myscript.py
```

Typical usage to run a program under control of the debugger is:

```
>>> import pdb
>>> import some_module
>>> pdb.run('some_module.function_to_text()')
> <string>(0)?()
(Pdb) c     # (c)ontinue
> <string>(1)?()
(Pdb) c     # (c)ontinue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

To get help, type **h** at the debugger prompt.

Stepping through a program

- s single-step, stepping into functions
- n single-step, stepping over functions
- r return from function
- c run to next breakpoint or end

The debugger provides several commands for stepping through a program. Use **s** to step through one line at a time, stepping into functions.

Use **n** to step over functions; use **r** to return from a function; use **c** to continue to next breakpoint or end of program.

Pressing Enter repeats most commands; if the previous command was list, the debugger lists the next set of lines.

Setting breakpoints

- Syntax

```
b list all breakpoints
b linenumber (, condition)
b file:linenumber (, condition)
b function name (, condition)
```

Breakpoints can be set with the b command. Specify a line number, or a function name, optionally preceded by the filename that contains it.

Any of the above can be followed by an expression (use comma to separate) to create a conditional breakpoint.

The **tbreak** command creates a one-time breakpoint that is deleted after it is hit the first time.

Profiling

- Use the **profile** module from the command line
- Shows where program spends the most time
- Output can be tweaked via options

Profiling is the technique of discovering the part of your code where your application spends the most time. It can help you find bottlenecks in your code that might be candidates for revision or refactoring.

To use the profiler, execute the following at the command line:

```
python -m profile scriptname.py
```

This will output a simple report to STDOUT. You can also specify an output file with the -o option, and the sort order with the -s option. See the docs for more information.

TIP The **pycallgraph2** module (third-party module) will create a graphical representation of an application's profile, indicating visually where the application is spending the most time.

Example

```
python -m profile count_with_dict.py
...script output...
         19 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       14    0.000    0.000    0.000    0.000 :0(get)
        1    0.000    0.000    0.000    0.000 :0(items)
        1    0.000    0.000    0.000    0.000 :0(open)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000 count_with_dict.py:3(<module>)
        1    0.000    0.000    0.000    0.000 profile:0(<code object <module> at
0xb74c36e0, file "count_with_dict.py", line 3>)
        0    0.000             0.000          profile:0(profiler)
```

Benchmarking

- Use the `timeit` module
- Create a `Timer` object with specified # of repetitions

Use the `timeit` module to benchmark two or more code snippets. To time code, create a `Timer` object, which takes two strings of code. The first is the code to test; the second is setup code, that is only run once per timer .

Call the `.timeit()` method with the number of times to call the test code, or call the `.repeat()` method which repeats `.timeit()` a specified number of times.

You can also use `timeit` from the command line. Use `-s` to specify startup code:

```
python -m timeit -s "startup code⋯" "code⋯."
```

Example

bm_range_vs_while.py

```
from timeit import Timer

setup_code = """
values = []
"""  # setup code is only executed once

test_code_one = '''
for i in range(10000):
    values.append(i)
values.clear()
'''  # code fragment executed many times

test_code_two = '''
i = 0
while i < 10000:
    values.append(i)
    i += 1
values.clear()
'''  # code fragment executed many times

t1 = Timer(test_code_one, setup_code)  # Timer object creates time-able code
t2 = Timer(test_code_two, setup_code)  # Timer object creates time-able code

print("test one:")
print(t1.timeit(1000))  # timeit() runs code fragment N times
print()

print("test two:")
print(t2.timeit(1000))  # timeit() runs code fragment N times
print()
```

bm_range_vs_while.py

```
test one:
0.518214814

test two:
1.124835274
```

Chapter 4 Exercises

Exercise 4-1

Pick several of your scripts (from class, or from real life) and run pylint on them.

Exercise 4-2

Use the builtin debugger or one included with your IDE to step through any of the scripts you have written so far.

Chapter 5: Logging

Objectives

- Configure application logging
- Modify log formats
- Log to alternate destinations
- Add logging to packages

Logging overview

- Main components
  - Loggers
  - Handlers
  - Filters
  - Formatters
- Several ways to configure

"Logging" means writing messages to report what your application is doing, so they can be examined later. This is especially important for troubleshooting and debugging.

The `logging` module provides tools to implement logging. It has several components that work together.

Loggers

Loggers are called by your code to send a log message. They are the only part directly used by your code, once logging is configured. Loggers pass the log message to a *handler*.

The default logger is called the 'root' logger.

Handlers

Handlers are responsible for getting the log messages to the correct destination, such as a file or `sys.stderr`. There are many handlers provided to handle files in different ways, as well as handlers for non-file destinations.

Formatters

Formatters are responsible for the layout of the log messages.

Filters

Filters are optional, but can be used control exactly which messages get output. They allow you to specify a subset of messages within a severity level.

Configuration

Configuration specifies logging details for a particular application. You can specify which loggers, handlers, and filters will be used, as well as the format of log messages, and many other details.

Simple Logging

- Minimum configuration
  - File name
  - Minimum level
- Call level-specific methods on `logging`
  - `logging.debug()`
  - `logging.info()`
  - `logging.warning()`
  - `logging.error()`
  - `logging.critical()`

For the very simplest approach, you can just import `logging` and start calling the per-level functions such as `logging.debug()`. This will use the minimum level of `logging.WARNING` and send the output to `sys.stderr`. In general, though, you will want to log to a file.

For simple file logging, configure the default logger with a file name and minimum logging level using `logging.basicConfig()`. Then call any of the per-level methods, such as `logging.debug()` or `logging.error()`, to output a log message for that level. If the message is at or above the minimal level, it will be added to the log file. This will configure the *root logger*, which is always used when you call `logging.error()` and friends.

If no file name is specified, or if `logging.basicConfig()` is not called and there is no other configuration, the message destination will be `sys.stderr`. If no minimum severity is specified, the default level will be `logging.WARNING`. Thus, with no configuration at all, messages at level `logging.WARNING` and above will be written to the console via `sys.stderr`.

By default, log files will continue to grow, and must be manually removed or truncated. If the file does not exist, it will be created. You could also use a different handler, which will manage the log files for you.

Although `logging.basicConfig()` is sufficient for simple cases, you may want to use dictionary-based configuration which can read data from a YAML or JSON file.

NOTE Loggers are thread-safe and async-safe.

Example

logging_really_simple.py

```python
import logging

# these will print to STDERR
logging.warning("I've got a bad feeling about this...")
logging.error("This is BAD")

# these won't print because default level is logging.WARNING
logging.info("The shortest president was James Madison")
logging.debug("I'm here!")
```

logging_really_simple.py

```
WARNING:root:I've got a bad feeling about this...
ERROR:root:This is BAD
```

Example

logging_simple.py

```python
import logging

logging.basicConfig(
    filename='../LOGS/simple.log',
    level=logging.WARNING,
)

logging.warning('This is a warning') # message will be output
logging.debug('This message is for debugging') # message will NOT be output
logging.error('This is an ERROR') # message will be output
logging.critical('This is ***CRITICAL***') # message will be output
logging.info('The capital of North Dakota is Bismark') # message will not be output
```

../LOGS/simple.log

```
WARNING:root:This is a warning
ERROR:root:This is an ERROR
CRITICAL:root:This is ***CRITICAL***
WARNING:root:This is a warning
ERROR:root:This is an ERROR
CRITICAL:root:This is ***CRITICAL***
```

Severity levels

The logger module provides 5 levels of message severity, from DEBUG to CRITICAL. When you set up a logger, you specify the minimum level of messages to be logged. For instance, if you set up the logger with the minimum level set to ERROR, then only messages at ERROR and CRITICAL levels will be logged. Setting the minimum level to DEBUG allows all messages to be logged.

Table 1. Logging Levels

Level	Value
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
UNSET	0

NOTE `logging.WARN` is a deprecated alias for `logging.WARNING`, and should not be used.

Table 2. Keyword arguments for `logging.basicConfig()`

Option	Default value	Description
`filename`	`None`	Name of log file[1]
`filemode`	`'a'`	mode of file to be opened (`'w'` will truncate log when app starts)
`format`	`'%(levelname)s %(name)s %(message)'`	Format for one log entry
`datefmt`	`None`	Format for the `(asctime)s` directive; uses same format as `time.strftime()`
`style`	`'%'`	Style for format string. Choices are `'%'`, `'{'`, or ``.
`level`	`logger.WARNING`	Set root logger level
`stream`	`sys.stderr`	Stream log messages will be written to (any writable file-like object)[1]
`handlers`	`None`	Iterable of handler objects[1]
`force`	`False`	Remove and close any existing handlers
`encoding`	`'utf-8'`	Encoding for log file
`errors`	`'backslashreplace'`	Value controls how encoding errors are handled.

[1] One, and only one, of `filename`, `stream` or `handlers` must be provided.

Formatting log entries

- Add `format=format` to `logging.basicConfig()` parameters
- Format is a string containing directives and other text

To format log entries, provide a format parameter to the `.logging.basicConfig()` method. This format will be a string contain special directives (i.e. placeholders) and, optionally, other text. The directives are replaced with logging information; other data is left as-is.

Directives are in the form `%(item)type`, where item is the data field, and *type* is the data type.

Formatting dates

The `%(asctime)s` directive can be formatted using the date format directives from `datetime.strftime`. Specify the `datefmt` option to `logging.basicConfig()`. === Formatting messages You can insert values into log messages by using the `%s` placeholder. When you call a log method such as `logging.warning()`, the first argument is the message, and any other arguments are inserted into the placeholders.

```
index = 12
try:
    print(mydata[index])
except IndexError as err:
    logging.error("Invalid index: %s", index)
```

Example

logging_formatted.py

```
import logging

logging.basicConfig(
    format='%(levelname)s %(name)s %(asctime)s %(filename)s %(lineno)d %(message)s', #
set the format for log entries
    datefmt="%x-%X",
    filename='../LOGS/formatted.log',
    level=logging.INFO,
)

logging.info("this is information")
logging.warning("this is a warning")
logging.error("this is an ERROR")
value = 38.7
logging.error("Invalid value %s", value)
logging.info("this is information")
logging.critical("this is critical")
```

../LOGS/formatted.log

```
INFO root 05/10/23-09:47:31 logging_formatted.py 11 this is information
WARNING root 05/10/23-09:47:31 logging_formatted.py 12 this is a warning
ERROR root 05/10/23-09:47:31 logging_formatted.py 13 this is an ERROR
ERROR root 05/10/23-09:47:31 logging_formatted.py 15 Invalid value 38.7
INFO root 05/10/23-09:47:31 logging_formatted.py 16 this is information
CRITICAL root 05/10/23-09:47:31 logging_formatted.py 17 this is critical
INFO root 05/10/23-09:47:31 logging_formatted.py 11 this is information
WARNING root 05/10/23-09:47:31 logging_formatted.py 12 this is a warning
ERROR root 05/10/23-09:47:31 logging_formatted.py 13 this is an ERROR
ERROR root 05/10/23-09:47:31 logging_formatted.py 15 Invalid value 38.7
INFO root 05/10/23-09:47:31 logging_formatted.py 16 this is information
CRITICAL root 05/10/23-09:47:31 logging_formatted.py 17 this is critical
```

Table 3. Log entry formatting directives

Directive	Description
`%(name)s`	Name of the logger (logging channel)
`%(levelno)s`	Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL)
`%(levelname)s`	Text logging level for the message ("DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL")
`%(pathname)s`	Full pathname of the source file where the logging call was issued (if available)
`%(filename)s`	Filename portion of pathname
`%(module)s`	Module (name portion of filename)
`%(lineno)d`	Source line number where the logging call was issued (if available)
`%(funcName)s`	Function name
`%(created)f`	Time when the LogRecord was created (time.time() return value)
`%(asctime)s`	Textual time when the LogRecord was created
`%(msecs)d`	Millisecond portion of the creation time
`%(relativeCreated)d`	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded (typically at application startup time)
`%(thread)d`	Thread ID (if available)
`%(threadName)s`	Thread name (if available)
`%(process)d`	Process ID (if available)
`%(message)s`	The result of record.getMessage(), computed just as the record is emitted

Table 4. Date Format Directives

Directive	Meaning
`%a`	Locale’s abbreviated weekday name
`%A`	Locale’s full weekday name
`%b`	Locale’s abbreviated month name
`%B`	Locale’s full month name
`%c`	Locale’s appropriate date and time representation
`%d`	Day of the month as a decimal number [01,31]
`%f`	Microsecond as a decimal number [0,999999], zero-padded on the left
`%H`	Hour (24-hour clock) as a decimal number [00,23]
`%I`	Hour (12-hour clock) as a decimal number [01,12]
`%j`	Day of the year as a decimal number [001,366]
`%m`	Month as a decimal number [01,12]
`%M`	Minute as a decimal number [00,59]
`%p`	Locale’s equivalent of either AM or PM.
`%S`	Second as a decimal number [00,61]
`%U`	Week number (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0
`%w`	Weekday as a decimal number [0(Sunday),6]
`%W`	Week number (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0
`%x`	Locale’s appropriate date representation
`%X`	Locale’s appropriate time representation
`%y`	Year without century as a decimal number [00,99]
`%Y`	Year with century as a decimal number
`%z`	UTC offset in the form +HHMM or -HHMM (empty string if the the object is naive)
`%Z`	Time zone name (empty string if the object is naive)
`%%`	A literal `'%'` character

Logging exception information

- Use logging.exception()
- Adds exception info to message
- Only in **except** blocks

The `logging.exception()` method will add exception information to the log message. It may only be called in an **except** block.

Example

logging_exception.py

```
import logging

logging.basicConfig( # configure logging
    filename='../LOGS/exception.log',
    level=logging.WARNING,  # minimum level
)

for i in range(3):
    try:
        result = i/0
    except ZeroDivisionError:
        logging.exception('Logging with exception info') # add exception info to the log
```

../LOGS/exception.log

```
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
11, in <module>
    result = i/0
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
11, in <module>
    result = i/0
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
11, in <module>
    result = i/0
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
11, in <module>
    result = i/0
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
11, in <module>
    result = i/0
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
11, in <module>
    result = i/0
ZeroDivisionError: division by zero
```

Loggers

- Calls logging methods
- Default logger is root
- Configuration is inherited from root *

The default logger is called 'root'. If you call the various methods to send log messages directly from logging, such as `logging.warn()`, they are called from the root logger. Using `logging.basicConfig()` configures the root logger. If you set a minimum severity level on the root logger, it is used by all loggers.

To get the root logger, call `logging.getLogger()` with no parameters.

For simple cases, you can just use the root logger, but for larger projects, you will probably want to create specific loggers, and configure each one as needed.

Each logger can have its own configuration for destination (file or other), severity level, message format, and other attributes.

Sometimes you might want to use more than one logger, to make it easy to filter out messages in different parts of your application.

You can call `logging.getLogger()` with any name to get a logger with that name. Then you can call the logging methods from that logger. You can create other loggers by calling `logging.getLogger()` with a name. A logger is really just a name, used for configuration purposes. `logging.getLogger()` returns a logger object from which you can call the message methods.

A logger is essentially just a name, so using multiple loggers lets you add distinctive information to the log message without changing anything else. You can use the module’s name plus a prefix or suffix.

Loggers are hierarchical, and the root logger is the implicit top of the hierarchy: the logger `spam.ham.toast` is really `root.spam.ham.toast`. Loggers inherit their configuration from root, and any parent loggers.

Example

logging_loggers.py

```
import logging

logging.basicConfig(
    format='%(levelname)s %(name)s %(asctime)s %(filename)s %(lineno)d %(message)s', #
format log entry
    datefmt="%Y:%M:%S::%X",  # set date/time format
    filename='../LOGS/loggers.log',
    level=logging.INFO,
)

logger_one = logging.getLogger(__name__ + "-1") # unique logger names
logger_two = logging.getLogger(__name__ + "-2")

logger_one.info("this is information")  # use logger 1
logger_two.warning("this is a warning")  # use logger 2
logger_one.info("this is information")
logger_two.critical("this is critical")
```

../LOGS/loggers.log

```
INFO __main__-1 2023:47:31::09:47:31 logging_loggers.py 14 this is information
WARNING __main__-2 2023:47:31::09:47:31 logging_loggers.py 15 this is a warning
INFO __main__-1 2023:47:31::09:47:31 logging_loggers.py 16 this is information
CRITICAL __main__-2 2023:47:31::09:47:31 logging_loggers.py 17 this is critical
INFO __main__-1 2023:47:32::09:47:32 logging_loggers.py 14 this is information
WARNING __main__-2 2023:47:32::09:47:32 logging_loggers.py 15 this is a warning
INFO __main__-1 2023:47:32::09:47:32 logging_loggers.py 16 this is information
CRITICAL __main__-2 2023:47:32::09:47:32 logging_loggers.py 17 this is critical
```

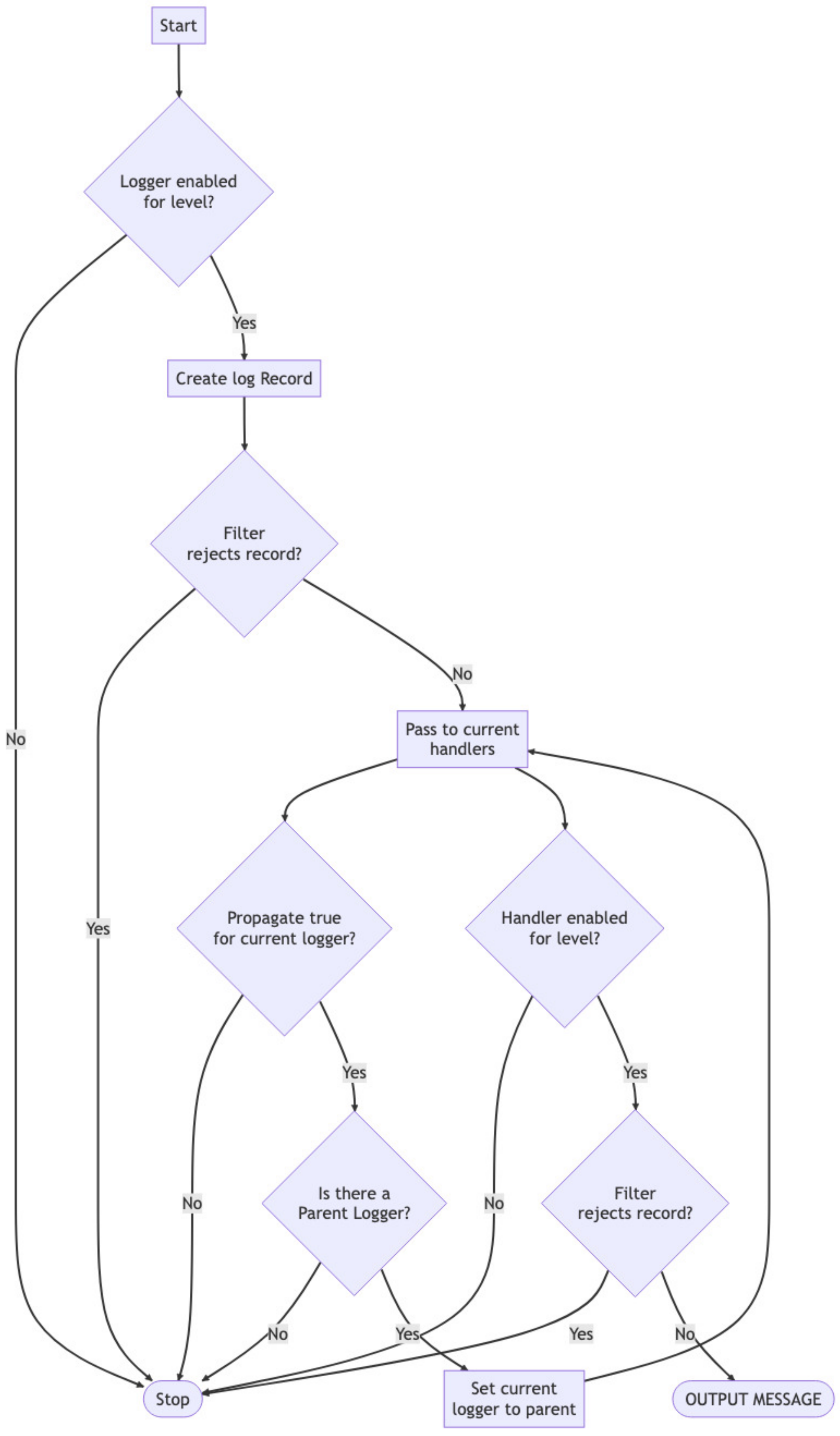

Start
Logger enabled for level?
Yes
No
Create log Record
Filter rejects record?
Yes
No
Pass to current handlers
Propagate true for current logger?
Handler enabled for level?
Yes
No
Is there a Parent Logger?
Yes
No
Filter rejects record?
No
Yes
Yes
No
Stop
Set current logger to parent
OUTPUT MESSAGE

Figure 1. Logging Flowchart

Logging from modules

- No configuration in module
- Create module-specific logger

When you create a module, it should be independent of any code that uses the module. This applies to logging as well.

To keep a module independent, create a logger with a distinctive name. Best practice is to use the builtin `__name__` variable. This will make the logger the same name as the module.

Loggers in modules inherit the root logger configuration.

Example

apple.py

```python
"""
Demonstrate logging from modules
"""
import logging
logger = logging.getLogger(__name__)


def apple():
    logger.warning("ambergris")
    logger.info("arrivederci")
    logger.error("aspiration")
    logger.debug("Function apple()")
```

banana.py

```
"""
Demonstrate logging from modules
"""
import logging
logger = logging.getLogger(__name__)

def banana():
    logger.warning("bravo")
    logger.info("Barbarella")
    logger.error("broccoli")
    logger.debug("banana()")
```

cherry.py

```
"""
Demonstrate logging from modules
"""
import logging
logger = logging.getLogger(__name__)

def cherry():
    logger.warning("ciao bello")
    logger.info("Cartesian")
    logger.error("cobblestones")
    logger.debug("Function cherry()")
```

logging_modules.py

```
import logging

logging.basicConfig(level=logging.DEBUG, filename="../LOGS/modules.log")

logger = logging.getLogger(__name__)

from apple import apple
from banana import banana
from cherry import cherry

def main():
    logger.info("Starting main script")
    apple()
    banana()
    cherry()
    logger.info("Ending main script")

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

../LOGS/modules.log

```
INFO:__main__:Starting main script
WARNING:apple:ambergris
INFO:apple:arrivederci
ERROR:apple:aspiration
DEBUG:apple:Function apple()
WARNING:banana:bravo
INFO:banana:Barbarella
ERROR:banana:broccoli
DEBUG:banana:banana()
WARNING:cherry:ciao bello
INFO:cherry:Cartesian
ERROR:cherry:cobblestones
DEBUG:cherry:Function cherry()
INFO:__main__:Ending main script
INFO:__main__:Starting main script
WARNING:apple:ambergris
INFO:apple:arrivederci
ERROR:apple:aspiration
DEBUG:apple:Function apple()
WARNING:banana:bravo
INFO:banana:Barbarella
ERROR:banana:broccoli
```

```
DEBUG:banana:banana()
WARNING:cherry:ciao bello
INFO:cherry:Cartesian
ERROR:cherry:cobblestones
DEBUG:cherry:Function cherry()
INFO:__main__:Ending main script
```

Multiple handlers

- Loggers can have one or more handlers
- Common handlers
  - `logging.FileHandler`
  - `logging.RotatingFileHandler`
  - `logging.StreamHandler`

Every logger can use any number of handlers. Many handlers are provided by `logging` and `logging.handlers`.

The `logging` module provides many log handlers. If you specify `filename=` in `logging.basicConfig()`, it uses `logging.FileHandler`; if you specify `stream=`, it uses `logging.StreamHandler`.

TIP If using `logging.basicConfig()`, you can assign an iterable of one or more handlers with the `handlers=` parameter.

Example

logging_multi_handlers.py

```
import logging

screen_handler = logging.StreamHandler()
screen_handler.setLevel(logging.ERROR)

file_handler = logging.FileHandler("../LOGS/multihandlers.log")
file_handler.setLevel(logging.INFO)

logging.basicConfig(
    handlers=[screen_handler, file_handler],
)

logging.info("starting multi handler demo")
logging.warning("this is a warning")
logging.error("I lost my hat")
logging.critical("we are out of coffee")
logging.info("ending multi handler demo")
```

../LOGS/multihandlers.log

```
WARNING:root:this is a warning
ERROR:root:I lost my hat
CRITICAL:root:we are out of coffee
WARNING:root:this is a warning
ERROR:root:I lost my hat
CRITICAL:root:we are out of coffee
```

Beyond console and file logging

- Use specialized handlers to write to other destinations
- Multiple handlers can be added to one logger
    - `NTEventLogHandler` Windows event log
    - `SysLogHandler` **syslog**
    - `SMTPHandler` log to email
    - `NullHandler` suppress messages

The `logging` module provides many preconfigured log handlers to send log messages to destinations other than a file or the console. Each handler has custom configuration appropriate to the destination.

Multiple handlers can be added to the same logger, so a log message could go to both a file and to email. Each handler can have its own minimum severity level. Thus, all messages could go to the message file, but only CRITICAL messages would go to email.

Be sure to read the documentation for the particular log handler you want to use

TIP https://docs.python.org/3/howto/logging-cookbook.html has many recipes for customizing logging.

NOTE On Windows, `altdest.py` requires elevated privileges. To do this, launch the windows Command Prompt or the Anaconda prompt as administrator. m

Table 5. Handlers included in `logging` or `logging.handlers`

Handler	Description
`StreamHandler`	send messages to streams or other *opened* file-like objects.
`FileHandler`	send messages to files.
`BaseRotatingHandler`	base class for handlers that rotate log files. Should not to be instantiated directly.
`RotatingFileHandler`	same as `FileHandler`, but supports maximum log file size and log file rotation.
`TimedRotatingFileHandler`	same as `FileHandler`, but rotates files at specified intervals
`SocketHandler`	send messages to TCP/IP sockets. [1]
`DatagramHandler`	send messages to UDP sockets. [1]
`SMTPHandler`	send messages to specified email address.
`SysLogHandler`	send messages to Unix local or remote syslog (Not supported on Windows)
`NTEventLogHandler`	send messages to Windows event log. (Windows only)
`MemoryHandler`	send messages to memory buffer (flushed on specific criteria).
`HTTPHandler`	send messages to HTTP server using GET or POST.
`WatchedFileHandler`	watch log file; when file changes, close and reopen file. (Not supported on Windows)
`QueueHandler`	send messages to queue (from queue or multiprocessing modules)
`NullHandler`	do nothing with error messages; used by library developers who want to avoid ‘No handlers could be found for logger XXX’ messages

[1] Unix domain sockets supported since 3.4

Example

logging_altdest.py

```python
import sys
from getpass import getpass
import logging
import logging.handlers

logger = logging.getLogger()  # get logger
logger.setLevel(logging.DEBUG)  # minimum log level

if sys.platform == 'win32':
    eventlog_handler = logging.handlers.NTEventLogHandler("Python Log Test")  # create NT event log handler
    logger.addHandler(eventlog_handler)  # install NT event handler
else:
    syslog_handler = logging.handlers.SysLogHandler()  # create syslog handler
    logger.addHandler(syslog_handler)  # install syslog handler

smtp_password = getpass("SMTP Password: ")

# note -- use your own SMTP server...
email_handler = logging.handlers.SMTPHandler(
    ("smtp2go.com", 2525),
    'LOGGER@pythonclass.com',
    ['jstrickler@gmail.com'],
    'Alternate Destination Log Demo',
    ('pythonclass', smtp_password),
)  # create email handler

logger.addHandler(email_handler)  # install email handler

logger.debug('this is debug')  # goes to all handlers
logger.critical('this is critical')  # goes to all handlers
logger.warning('this is a warning')  # goes to all handlers
```

(no output to file or console)

Example

logging_null.py

```python
import logging

logging.basicConfig(
    # filename='../LOGS/simple.log',
    level=logging.WARNING,
    handlers=[logging.NullHandler()],
)

logging.warning('This is a warning') # message will be output
logging.debug('This message is for debugging') # message will NOT be output
logging.error('This is an ERROR') # message will be output
logging.critical('This is ***CRITICAL***') # message will be output
logging.info('The capital of North Dakota is Bismark') # message will not be output
```

(no output to file or console)

Configuration

- Many approaches
  - `logging.basicConfig()`
  - `logging.dictConfig()`
  - `logging.fileConfig()`
  - *Ad hoc* configuration

In addition to `logging.basicConfig()`, you can configure logging via a dictionary or file.

Dictionary-based configuration

For dictionary-based configuration, use `logging.dictConfig()`, and pass in a dictionary whose keys are the same configuration options as `logging.basicConfig()`. There are a few differences. For severity levels, use `"ERROR"` instead of `logging.ERROR`. You can configure each logger by its name. The configuration version must be specified. This refers to the `logging` dictionary configuration schema, and is currently set to 1. In the future, if the schema should change, this will provide backwards compatibility.

This is considered best practice for logging configuration. If you want to keep the configuration in a file, you can store the configuration in a YAML or JSON file and read it into a Python dictionary.

NOTE `logging_config_yaml.py` and `logging_config_json.py` in the EXAMPLES folder show how to load a dictionary from a file.

Ad hoc configuration

You can also update logger and handler attributes on-the-fly.

Example

logging_config_dict.py

```python
from argparse import ArgumentParser
import logging
from logging.config import dictConfig

# set up -d option to script
parser = ArgumentParser(description="Logging config demo")
parser.add_argument("-d", dest="debug", action="store_true", help="show debug messages")
args = parser.parse_args()

CONFIG = {
    "version": 1,  # required
    "loggers": {
        "root": {  # root can also be directly configured
            "level": 'ERROR',
            "handlers": ["console", "file"],  # handler IDs from this file
        },
    },
    "formatters": {
        "minimal": {
            "format": '%(name)s %(message)s'
        },
        "full": {
            "format": '%(asctime)s %(levelname)-10s %(name)-10s %(message)s',
            "datefmt": "%x %X",
        },
    },
    "handlers": {
        "console": {
            "class": "logging.StreamHandler",  # handler must be a string
            "formatter": "minimal",  # formatter ID from this file
            "level": "ERROR",    # translated to logging.ERROR
            "stream": "ext://sys.stderr",  # default is sys.stderr
        },
        "file": {
            "class": "logging.FileHandler",
            "formatter": "full",
            "filename": "../LOGS/dictconfig.log",
            "level": "DEBUG",
        },
    },
}

dictConfig(CONFIG)
```

```
if args.debug:
    root = logging.getLogger()
    # print(root.handlers)
    root.setLevel(logging.DEBUG)  # change level of file handler

logging.info("script begins")
logging.debug("configuration from a dictionary")
logging.error("Whoa -- this is not good")
logging.critical("Dude, we're doomed")
logging.info("script ends")
```

logging_config_dict.py

```
root Whoa -- this is not good
root Dude, we're doomed
```

../LOGS/dictconfig.log

```
05/10/23 09:47:32 ERROR      root       Whoa -- this is not good
05/10/23 09:47:32 CRITICAL   root       Dude, we're doomed
05/10/23 09:47:32 ERROR      root       Whoa -- this is not good
05/10/23 09:47:32 CRITICAL   root       Dude, we're doomed
```

logging_config_dict.py -d

```
root Whoa -- this is not good
root Dude, we're doomed
```

../LOGS/dictconfig.log

```
05/10/23 09:47:32 ERROR      root       Whoa -- this is not good
05/10/23 09:47:32 CRITICAL   root       Dude, we're doomed
05/10/23 09:47:32 ERROR      root       Whoa -- this is not good
05/10/23 09:47:32 CRITICAL   root       Dude, we're doomed
05/10/23 09:47:32 INFO       root       script begins
05/10/23 09:47:32 DEBUG      root       configuration from a dictionary
05/10/23 09:47:32 ERROR      root       Whoa -- this is not good
05/10/23 09:47:32 CRITICAL   root       Dude, we're doomed
05/10/23 09:47:32 INFO       root       script ends
05/10/23 09:47:32 INFO       root       script begins
05/10/23 09:47:32 DEBUG      root       configuration from a dictionary
```

```
05/10/23 09:47:32 ERROR      root       Whoa -- this is not good
05/10/23 09:47:32 CRITICAL   root       Dude, we're doomed
05/10/23 09:47:32 INFO       root       script ends
```

File-based configuration

- Older method
- Not as flexible as dict config
- Quirks and disadvantages

To use configuration from a file, the format is similar to Windows `.INI` files. File configuration has been around longer than dictionary configuration, and there are some quirks and disadvantages, without any real advantages.

The entries in the file correspond to the `logging.basicConfig()` parameters and the keys used with `logging.dictConfig()`. The root logger must be configured with the `[logger_root]` entry.

In general, the best way to configure Python logging for non-trivial cases is to put the configuration information in a JSON or YAML file and read it into a dictionary, which is then passed to `logging.dictConfig()`.

Disadvantages

- No comments are allowed in the file
- Keys must be defined and then joined to "object type" names
- No indentation of nested information

Example

logging.conf

```
[loggers]
keys=root

[logger_root]
handlers=console,file
level=ERROR

[formatters]
keys=minimal,full

[formatter_minimal]
format=%(name)s %(message)s

[formatter_full]
format=%(asctime)s %(levelname)-10s %(name)-10s %(message)s
datefmt=%x %X

[handlers]
keys=console,file

[handler_console]
class=StreamHandler
formatter=minimal
level=ERROR
stream=ext://sys.stderr

[handler_file]
class=FileHandler
formatter=full
args=('../LOGS/fileconfig.log',)
level=INFO
```

logging_config_file.py

```
from argparse import ArgumentParser
import logging
from logging.config import fileConfig

# set up -d option to script
parser = ArgumentParser(description="Logging config demo")
parser.add_argument("-d", dest="debug", action="store_true", help="show debug messages")
args = parser.parse_args()

fileConfig('logging.conf')

if args.debug:
    root = logging.getLogger()
    # print(root.handlers)
    root.setLevel(logging.DEBUG)  # change level of file handler

logging.info("script begins")
logging.debug("configuration from a dictionary")
logging.error("Whoa -- this is not good")
logging.critical("Dude, we're doomed")
logging.info("script ends")
```

logging_config_file.py

```
root Whoa -- this is not good
root Dude, we're doomed
```

../LOGS/fileconfig.log

```
05/10/23 09:47:32 ERROR      root       Whoa -- this is not good
05/10/23 09:47:32 CRITICAL   root       Dude, we're doomed
05/10/23 09:47:32 ERROR      root       Whoa -- this is not good
05/10/23 09:47:32 CRITICAL   root       Dude, we're doomed
```

logging_config_file.py -d

```
root Whoa -- this is not good
root Dude, we're doomed
```

../LOGS/fileconfig.log

```
05/10/23 09:47:32 ERROR      root       Whoa -- this is not good
05/10/23 09:47:32 CRITICAL   root       Dude, we're doomed
05/10/23 09:47:32 ERROR      root       Whoa -- this is not good
05/10/23 09:47:32 CRITICAL   root       Dude, we're doomed
05/10/23 09:47:32 INFO       root       script begins
05/10/23 09:47:32 ERROR      root       Whoa -- this is not good
05/10/23 09:47:32 CRITICAL   root       Dude, we're doomed
05/10/23 09:47:32 INFO       root       script ends
05/10/23 09:47:32 INFO       root       script begins
05/10/23 09:47:32 ERROR      root       Whoa -- this is not good
05/10/23 09:47:32 CRITICAL   root       Dude, we're doomed
05/10/23 09:47:32 INFO       root       script ends
```

Chapter 5 Exercises

Exercise 5-1 (president_log.py, pres_log_main.py)

Modify the existing module `president.py` to add logging. Add a critical message if the `presidents.txt` file is not found, and a warning message if an invalid term number is passed to the constructor. This should be done with `try-except` blocks.

Write a separate script to import the module and create one or more President instances. Try to create an instance with an invalid term number. Your module should log the error. Try temporarily changing the data file script name and see if you module correctly logs the error. The script should log when it starts and stops at `logging.INFO` level.

Chapter 6: Type Hinting

Objectives

- Learn how to annotate variables with their type
- Understand what the type hints do **not** provide
- Employ the `typing` module to annotate collections
- Use the `Union` and `Optional` types correctly
- Write stub interfaces using type hints

Type Hinting

Python supports optional type hinting of variables, functions, and parameters. **This is not enforced by the Python interpreter.**

Types may be specified with the declaration of a variable:

```
count: int = 0
```

Once again, this is type **hinting** only; the Python interpreter does not check the types in any way:

```
# Valid Python, incorrect intent
valid: dict = (3, 'hello')
```

Python functions use a `->` to indicate a return type; function parameters are annotated with type information in the same way as any variable.

```
def shout(word: str, times: int = 1) -> str:
    return word.upper() * times
```

Argument lists and keyword argument lists may be type-hinted, the values are then expected to uniformly be of that type. For keyword arguments, the keys are still strings; only the values get the type hint.

```
def shout_various(*, **kwargs: int) -> None:
    for word in kwargs:
        print(word.upper() * kwargs[word])
```

Static Analysis Tools

If these type hints are not used by the Python interpreter, how are they useful? While Python (currently) does not use the type hints in any way, static analysis tools do. The most common tool for static type analysis is the `mypy` module. This is not part of the standard distribution and is a separate `pip` install.

```
> pip install mypy
> python3 -m mypy multishout.py
```

The `mypy` module will scan the code (technically, AST of the code) and try to determine, at "compile" time, whether the types expected and used match up correctly. It will emit errors when it detects static typing problems in the code.

`mypy` even supports scanning the inline arbitrary code that may be present in a format-string literal.

```
word: str = 'hello'
# mypy will report an error on the next line
print(f'{word + 3})
```

`mypy` will emit errors, warnings, and notes of what it finds. While the output is quite configurable, most projects would benefit from fixing any and all issues found by `mypy`.

Runtime Analysis Tools

In theory, Python's type hints can be checked at runtime. Unfortunately, such checking involves a very high performance cost.

`__annotations__`

The mechanism behind how type hints are tracked is through a dictionary called `__annotations__`. There is one at the global level, for any global variables declared with type hinting. There also exist `__annotations__` dictionary attributes on both functions and classes. (Notably, while such annotations may be made on local variables, Python does not track these programmatically; there is no local `__annotations__` dictionary. Static tools such as `mypy` are expected to parse these annotations themselves, rather than having the runtime cost of the interpreter parsing them and filling in another dictionary.)

This `__annotations__` dictionary actually just stores key: value pairs of the string version of the variable name, and the type specified. Interestingly, the "type" could be any Python value!

```
def weird(word: len, times: 0 = 1) -> 'unk':
    return word * times
```

This is perfectly valid Python code. The `__annotations__` dictionary stores the value of each annotation.

```
>>> weird.__annotations__
{'word': <built-in function len>, 'times': 0, 'return': 'unk'}
```

The return type of the function is stored in the key `return`. This ensures that no parameter name will conflict with the return type (as `return` is a keyword).

Forward References

Not all types may be available at the time that a given piece of Python code is compiled to bytecode. In other words, **forward references**, where a type is referred to before it is defined, are needed. The standard way to do this in Python is by using strings; tools are expected to handle this forward reference.

```
class First:
    ...
    # The type Second is not yet available
    # to python, so it must be
    # forward-declared using a string
    def process(self, item: 'Second') -> str:
        ...

class Second:
    ...
    # The type First is available to
    # python, so it can just reference
    # the First symbol directly
    def create(self, data: First) -> str:
        ...
```

Other languages use similar concepts for declaring a type without defining it. The `mypy` tool deals correctly with these forward references.

Forward references are also how various operator overloads may need to be written, to refer to the current class.

```
class Matrix:
    def __matmul__(self, obj: 'Matrix') -> 'Matrix':
        ...
```

Forward references can also just be part of a type hint, they need not "gobble up" the entire type hint.

```
class Tree:
    def leaves(self) -> List['Tree']:
        ...
```

`typing` Module

The `typing` module makes it easier to refer to containers as a kind of type in Python code. To declare that a tuple has specific types of fields:

```python
from typing import Tuple

# Expects a three-tuple with types of str, int, and float
def process(record: Tuple[str, int, float]) -> None:
    ...
```

The `typing` module makes available the many type-wrapper classes. See https://docs.python.org/3/library/typing.html for the complete list.

Input Types

Most input types should not be of type List, but rather how the list is used, so either an Iterable or a Collection. Most of the more-specific containers (such as Dict, Set, and List) should generally only find use as return values.

Tuple objects generally specify exactly which type each positional value is, such as Tuple[str, int, str]. Tuples of arbitrary length (but the same type throughout) may be specified using the Ellipsis object.

```
def ziptuple(words: Tuple[str, ...], times: Tuple[int, ...]) -> Generator[str, None,
None]:
    for s, i in zip(words, times):
        yield s * i
```

The typing.Generator type takes exactly three types for its template: the type yielded, the type sent, and the type returned by the generator. If any of those types are not used, they should be set to None. Unusually, the send-type for a generator is contravariant (because a generator is a function).

```
def primes() -> Generator[int, None, None]:
    yield 2
    yield 3
    yield 5

def process() -> Generator[int, Manager, None]:
    value = 0
    while True:
        sent = yield value
        value = sent.employee_id

results = process();
for c in results:
    x = lookup_employee(c.boss)
    # Valid,
    results.send(x)
```

Creating Types

Creating new type aliases in Python follows the same process as creating any other kind of variable in Python.

```
FrequencyDict = Dict[str, int]
LangMap = Dict[str, FrequencyDict]

def determine_language(d: LangMap) -> None:
    ...
```

When working with generic types, it can be important to signal that a given type is maintained throughout a function call. Consider a function that filters a value from a container:

```
from typing import TypeVar, Sequence

Choice = TypeVar('Choice')

def third(coll: Sequence[Choice]) -> Choice:
    return coll[2]
```

In this case, a generic type is needed (rather than inheriting from a specific type), so a new one is created with `TypeVar`. This class can restrict itself to a specific set of types:

```
from typing import TypeVar, Sequence

Numeric = TypeVar('Numeric', int, float)

def first(coll: Sequence[Numeric]) -> Numeric:
    return coll[0]
```

In addition, `TypeVar` also allows for fine-grained control of the variance of a given container's type.

Variance

When dealing with types and inheritance, certain interactions between things that contain a type need to be made explicit. For instance, given a relationship of `Cat`, `Mammal`, `Animal`, and `Dog`, consider the following method:

```
def brush_hair(a: Mammal): ...
```

It makes logical sense that any subtype of `Mammal` would be able to have its hair brushed. So, passing a subtype to `brush_hair` should be acceptable to the type system. This kind of relationship between an argument's type and its inheritance is known as **covariance**: the subtype of an argument can be used in the place of the argument's type. This is also known as the **Liskov Substitution Principle**. But a function with a different signature involves subtle traps.

```
def brush_hair_all(a: List[Mammal]) -> None: ...

def add_cat(a: List[Mammal]) -> None: ...
```

If a function existed that would brush the hair of a whole collection of `Mammal`s, it makes sense that there might be both `Cat` and `Dog` objects in the collection, all of which get their hair brushed. Similarly, a `List[Cat]` collection could be passed to the function. This function is covariant.

But, suppose a function with an identical signature existed, `add_cat`. It takes a collection of `Mammal` objects and adds a cat, which at first glance seems acceptable. The problem is that this function could have been passed a `List[Dog]` collection, and adding a `Cat` to it would violate its type! Some languages handle this at runtime, even though it can be discovered statically.

On the other hand, if a `List[Mammal]` or `List[Animal]` collection was passed, the `add_cat` function would work perfectly well. Note that these are both supertypes, rather than subtypes. That means that the `add_cat` function is **contravariant**: the supertype of an argument can be used in place of the argument's type. This form of variance can be noted expressly in Python's type-hinting system.

```
T_co = TypeVar('T_co', covariant=True)
T_contra = TypeVar('T_contra', contravariant=True)

class ListOrMoreSpecific(List[T_co]): pass
class ListOrLessSpecific(List[T_contra]): pass

def brush_hair_all(a: ListOrMoreSpecific[Mammal]) -> None:
    for x in a:
        print(f'Pet the {x.hair()}')

def add_cat(a: ListOrLessSpecific[Cat]) -> None:
    a.append(Cat())
```

The default variance of the collections in the `typing` module are **invariant**, which means that only the exact type specified is permitted.

`Union` and `Optional`

Python's type system is actually quite robust from a theoretical perspective, allowing a form of tagged unions and optional types.

`Union` Types

A `Union` type is allowed to be one of a number of possible types. For instance, a value that may be either a `float` or a `str` may be written as:

```
apartment: Union[int,str]
if apartment.isinstance(int):
    print(f'Apartment #{apartment}')
else:
    print(f'Apartment {apartment}')
```

`Union` s may specify any number of valid types. It is the job of the calling code to tease out the correct type if they must be treated differently.

```
def destroy(junk: Union[Car,Refrigerator,ACUnit]) -> None:
if junk.isinstance(Refrigerator):
    junk.remove_door()
elif junk.isinstance(Car):
    crush(junk)
else:
    junk.drain_freon()
```

`Optional` Types

An important specific case of `Union` types is the `Optional` type. An `Optional` type is one that is either `None`, or a specified type.

```
def get_record(id: int) -> Optional[Record]:
    row = db.query('WHERE id = ?', id)
    if not row:
        return None
    else:
        return row
```

With Python's support for exceptions, this may seem unusual. But, there are cases where a value may be present, or absent. The `Optional` type is excellent for type-checking these cases, as `mypy` will detect if the wrapped type is being used without a branch for checking the `None` possibility.

```
def annoy_cat(times: Optional[int]) -> str:
# This line generates the mypy output:
#note: Right operand is of type "Optional[int]"
    return 'meow' * times
```

This allows for dealing with detectable default values in a type-safe way.

```
# print the phrase some number of times, unless the number is not specified
def print_times(phrase: str, times: Optional[int] = None) -> None:
    if times is None:
        print(str)
    else:
        print(str * times)
```

multimethod and functools.singledispatch

One of the utility of types in a programming language is the ability to pick a specific function or method based on its arguments' types. Baked into the standard library is the `functools.singledispatch` decorator. This involves decorating a function, and then defining multiple implementations with different types for the **first** argument.

```
@functools.singledispatch
def wound(x):
    print(f'Hit the {x}')

@wound.register
def _(x: Twistable):
    print(f'Wind the {x}')
```

Note that any additional registered functions must **not** share the same function or method name as the originally decorated function, unlike the `@property` decorator.

The functions are typed **only** on the basis of the first argument. Having a different number of arguments, or differently-typed later arguments, will not be used by `@singledispatch`.

multimethod

`multimethod` is a third-party library that provides even greater support for multiple dispatch functions. It provides a decorator, `@multimethod`, that takes into account the types of all function arguments.

```
@multimethod
def drop_bass(loc: Location, m: Music):
    print('WUBWUBWUBWUBWUBWUB DRRRRRR')
    ...

@multimethod
def drop_bass(loc: Location, f: Fish):
    print('Sploosh')
    ...
```

Further, the `@multimethod` decorator may be passed the keyword `strict`, which will raise an exception if no function signature can be found with the passed-in types.

```
@multimethod(strict=True)
def drop_bass(loc: Location, f: Fish):
    print('Sploosh')
    ...

b = StringBass()
# Raises TypeError
drop_bass(concert, b)
```

Stub Type Hinting

While type hinting can be a useful tool, it can be limiting when integrating with other source code. If a third-party library does not use type hints, then the type errors generated by that module may not be correctable in the current project.

The mypy utility can actually read a separate file to find out the type interface of a different module.

```
$ export MYPYPATH=$VIRTUAL_ENV/stubs
```

This path-setting may be best in a given virtual environment, so that different versions of a given library can have appropriate type hinting. In the stub directory, create a Python interface (.pyi) with the same name as the module to annotate. Then fill the Python interface file with an outline of the public types of the module to annotate.

```
class Card:
    suit: str
    rank: int
    def __repr__(self) -> str: ...

    def deal(deck: 'Deck', players: int = ...) -> Tuple[Deck]: ...
```

In this code sample, the ellipses are **literal** ellipses. The Ellipsis singleton is used by mypy to allow Python to parse the interface without running any code.

Now, mypy is able to parse this interface file to determine the correct annotated types, and can perform static analysis on code using the referenced module.

Chapter 6 Exercises

Exercise 6-1 (grep_sed.py)

Write type hints for the following code, rewriting the code as necessary:

```python
def grep(x):
    '''Only emits lines of text that contain x'''
    while True:
        coll = yield
        if x in coll:
            yield x

def sed(pattern, replacement):
    '''Replace any lines containing pattern with replacement'''
    while True:
        value = yield
        if pattern in value:
            yield replacement
```

Exercise 6-2 (sow.py)

Write an overloaded function `sow` that behaves differently when it is passed a `Pig` object versus a `Seed` object.

Exercise 6-3 (append_42.py)

Write and annotate a function `append_42` that appends a `42` to the argument, a list. If no list is supplied, a new one should be created and returned.

Exercise 6-4 (double_arg_deco.py)

Write a decorator `@double` that will double all numerically-annotated arguments passed to the function it decorates.

Chapter 7: Unit Testing with pytest

Objectives

- Understand the purpose of unit tests
- Design and implement unit tests with pytest
- Run tests in different ways
- Use builtin fixtures
- Create and use custom fixtures
- Mark tests for running in groups
- Learn how to mock data for tests

What is a unit test?

- Tests *unit* of code in isolation
- Ensures repeatable results
- Asserts expected behavior

A *unit test* is a test which asserts that an isolated piece of code (one function, method, class, or module) has some expected behavior. It is a way of making sure that code provides repeatable results.

There are four main components of a unit testing system:

1. Unit tests – individual assertions that an expected condition has been met
2. Test cases – collections of related unit tests
3. Fixtures — provide data to set up tests in order to get repeatable results
4. Test runners – utilities to execute the tests in one or more test cases

Unit tests should each test one aspect of your code, and each test should be independent of all other tests, including the order in which tests are run.

Each test asserts that some condition is true.

Unit tests may collected into a **test case**, which is a related group of unit tests. With **pytest**, a test case can be either a module or a class.

Fixtures provide repeatable, known input to a test.

The final component is a **Test runner**, which executes one, some, or all tests and reports on the results. There are many different test runners for pytest. The builtin runner is very flexible.

The pytest module

- Provides
    - test runner
    - fixtures
    - special assertions
    - extra tools
- Not based on xUnit[1]

The **pytest** module provides tools for creating, running, and managing unit tests.

Each test supplies one or more **assertions**. An assertion confirms that some condition is true.

Here's how **pytest** implements the main components of unit testing:

unit test
: A normal Python function that uses the **assert** statement to assert some condition is true

test case
: A class *or* a module than contains unit tests (tests can be grouped with *markers*).

fixture
: A special parameter of a unit test function that provides test resources (fixtures can be nested).

test runner
: A text-based test runner is built in, and there are many third-party test runners

pytest is more flexible than classic **xUnit** implementations. For example, fixtures can be associated with any number of individual tests, or with a test class. Test cases need not be classes.

[1] The builtin unit testing module, **unittest**, *is* based on **xUnit** patterns, as implemented in Java and other languages.

Creating tests

- Create test functions
- Use builtin **assert**
- Confirm something is true
- Optional message

To create a test, create a function whose name begins with "test". These should normally be in a separate script, whose name begins with "test_" or ends with "_test". For the simplest cases, tests do not even need to import **pytest**.

Each test function should use the builtin **assert** statement one or more times to confirm that the test passes. If the assertion fails, the test fails.

pytest will print an appropriate message by introspecting the expression, or you can add your own message after the expression, separated by a comma

It is a good idea to make test names verbose. This will help when running tests in verbose mode, so you can see what tests are passing (or failing).

```
assert result == 'spam'
assert 2 == 3, "Two is not equal to three!"
```

Example

pytests/test_simple.py

```
def test_two_plus_two_equals_four():  # tests should begin with "test" (or will not be
found automatically)
    assert 2 + 2 == 4   # if assert statement succeeds, the test passes
```

Running tests (basics)

- Needs a test runner
- **pytest** provides ***pytest*** script

To actually run tests, you need a *test runner*. A test runner is software that runs one or more tests and reports the results.

pytest provides a script (also named **pytest**) to run tests.

You can run a single test, a test case, a module, or all tests in a folder and all its subfolders.

```
pytest test_…py
```

to run the tests in a particular module, and

```
pytest -v test_…py
```

to add verbose output.

By default, pytest captures (and does not display) anything written to stdout/stderr. If you want to see the output of **print()** statements in your tests, add the **-s** option, which turns off output capture.

```
pytest -s …
```

NOTE In older versions of pytest, the test runner script was named **py.test**. While newer versions support that name, the developers recommend only using **pytest**.

TIP PyCharm automatically detects a script containing test cases. When you run the script the first time, PyCharm will ask whether you want to run it normally or use its builtin test runner. Use **Edit Configurations** to modify how the script is run. Note: in PyCharm's settings, you can select the default test runner to be **pytest**, **Unittest**, or other test runners.

Special assertions

- Special cases
  - pytest.raises()
  - pytest.approx()

There are two special cases not easily handled by **assert**.

pytest.raises

For testing whether an exception is raised, use **pytest.raises()**. This should be used with the **with** statement:

```
with pytest.raises(ValueError):
    w = Wombat('blah')
```

The assertion will succeed if the code inside the **with** block raises the specified error.

pytest.approx

For testing whether two floating point numbers are *close enough* to each other, use **pytest.approx()**:

```
assert result == pytest.approx(1.55)
```

The default tolerance is 1e-6 (one part in a million). You can specify the relative or absolute tolerance to any degree. Infinity and NaN are special cases. NaN is normally not equal to anything, even itself, but you can specify `nanok=True` as an argument to approx().

NOTE See https://docs.pytest.org/en/latest/reference.html#pytest-approx for more information on pytest.approx()

Example

pytests/test_special_assertions.py

```
import pytest
import math

FILE_NAME = 'IDONOTEXIST.txt'

def test_missing_filename():
    with pytest.raises(FileNotFoundError):  # assert FileNotFoundError is raised inside
block
        open(FILE_NAME)  # will fail test if file is not found

def test_list():
    print()
    assert (.1 + .2) == pytest.approx(.3)  # fail unless values are within 0.000001 of
each other (actual result is 0.30000000000000004)

def test_approximate_pi():
    assert 22 / 7 == pytest.approx(math.pi, .001)  # Default tolerance is 0.000001;
smaller (or larger) tolerance can be specified
```

Fixtures

- Provide resources for tests
- Implement as functions
- Scope
  - Per test
  - Per class
  - Per module
- Source of fixtures
  - Builtin
  - User-defined

When writing tests for a particular object, many tests might require an instance of the object. This instance might be created with a particular set of arguments.

What happens if twenty different tests instantiate a particular object, and the object's API changes? Now you have to make changes in twenty different places.

To avoid duplicating code across many tests, pytest supports *fixtures*, which are functions that provide information to tests. The same fixture can be used by many tests, which lets you keep the fixture creation in a single place.

A fixture provides items needed by a test, such as data, functions, or class instances. A fixtures can be either builtin or custom.

Fixtures provide * Consistency — test uses the same, repeatable data * Readability — keeps test itself short and simple * Auto-use — Reduces number of imports * Teardown — Provides cleanup capabilities

TIP Use `py.test --fixtures` to list all available builtin and user-defined fixtures.

User-defined fixtures

- Decorate with **pytest.fixture**
- Return value to be used in test
- Fixtures may be nested

To create a fixture, decorate a function with **pytest.fixture**. Whatever the function returns is the value of the fixture.

To use the fixture, pass it to the test function as a parameter. The return value of the fixture will be available as a local variable in the test.

Fixtures can take other fixtures as parameters as well, so they can be nested to any level.

It is convenient to put fixtures into a separate module so they can be shared across multiple test scripts.

TIP Add docstrings to your fixtures and the docstrings will be displayed via `pytest --fixtures`

Example

pytests/test_simple_fixture.py

```python
from collections import namedtuple
import pytest
import sqlite3

Person = namedtuple('Person', 'first_name last_name')  # create object to test

FIRST_NAME = "Guido"
LAST_NAME = "Von Rossum"

db_conn = sqlite3.connect('../../DATA/presidents.db')
db_cursor = db_conn.cursor()
db_cursor.row_factory = sqlite3.Row  # set the row factory to be a Row object

@pytest.fixture

def presidents():
    db_cursor.execute('select * from presidents')
    return db_cursor.fetchall()

@pytest.fixture  # mark person as a fixture
def person():
    """
    Return a 'Person' named tuple with fields 'first_name' and 'last_name'
    """
    return Person(FIRST_NAME, LAST_NAME)  # return value of fixture

def test_first_name(person):  # pass fixture as test parameter
    assert person.first_name == FIRST_NAME

def test_last_name(person):  # pass fixture as test parameter
    assert person.last_name == LAST_NAME

def test_john_tyler_is_from_virginia(presidents):
    assert presidents[9]['birthstate'] == 'Virginia'  # John Tyler is 10th president
```

Builtin fixtures

- Variety of common fixtures
- Provide
  - Temp files and dirs
  - Logging
  - STDOUT/STDERR capture
  - Monkeypatching tools

Pytest provides a large number of builtin fixtures for common testing requirements.

Using a builtin fixture is like using user-defined fixtures. Just specify the fixture name as a parameter to the test. No imports are needed for this.

See https://docs.pytest.org/en/latest/reference.html#fixtures for details on builtin fixtures.

Example

pytests/test_builtin_fixtures.py

```python
COUNTER_KEY = 'test_cache/counter'

def test_cache(cache):  # cache persists values between test runs
    value = cache.get(COUNTER_KEY, 0)
    print("Counter before:", value)
    cache.set(COUNTER_KEY, value + 1)  # cache fixture is similar to dictionary, but with
.set() and .get() methods
    value = cache.get(COUNTER_KEY, 0)  # cache fixture is similar to dictionary, but with
.set() and .get() methods
    print("Counter after:", value)
    assert True   # Make test successful

def hello():
    print("Hello, pytesting world")

def test_capsys(capsys):
    hello()  # Call function that writes text to STDOUT
    out, err = capsys.readouterr()  # Get captured output
    print("STDOUT:", out)

def bhello():
    print(b"Hello, binary pytesting world\n")

def test_capsysbinary(capsys):
    bhello()  # Call function that writes binary text to STDOUT
    out, err = capsys.readouterr()  # Get captured output
    print("BINARY STDOUT:", out)

def test_temp_dir1(tmpdir):
    print("TEMP DIR:", str(tmpdir))  # tmpdir fixture provides unique temporary folder
name

def test_temp_dir2(tmpdir):
    print("TEMP DIR:", str(tmpdir))

def test_temp_dir3(tmpdir):
    print("TEMP DIR:", str(tmpdir))
```

Table 6. Pytest Builtin Fixtures

Fixture	Brief Description
`cache`	Return cache object to persist state between testing sessions.
`capsys`	Enable capturing of writes (text mode) to `sys.stdout` and `sys.stderr`
`capsysbinary`	Enable capturing of writes (binary mode) to `sys.stdout` and `sys.stderr`
`capfd`	Enable capturing of writes (text mode) to file descriptors 1 and 2
`capfdbinary`	Enable capturing of writes (binary mode) to file descriptors 1 and 2
`doctest_namespace`	Return `dict` that will be injected into namespace of doctests
`pytestconfig`	Session-scoped fixture that returns `_pytest.config.Config` object.
`record_property`	Add extra properties to the calling test.
`record_xml_attribute`	Add extra xml attributes to the tag for the calling test.
`caplog`	Access and control log capturing.
`monkeypatch`	Return `monkeypatch` fixture providing monkeypatching tools
`recwarn`	Return `WarningsRecorder` instance that records all warnings emitted by test functions.
`tmp_path`	Return `pathlib.Path` instance with unique temp directory
`tmp_path_factory`	Return a `_pytest.tmpdir.TempPathFactory` instance for the test session.
`tmpdir`	Return `py.path.local` instance unique to each test
`tmpdir_factory`	Return `TempdirFactory` instance for the test session.

Configuring fixtures

- Create **conftest.py**
- Automatically included
- Provides
    - Fixtures
    - Hooks
    - Plugins
- Directory scope

The **conftest.py** file can be used to contain user-defined fixtures, as well as hooks and plugins. Subfolders can have their own conftest.py, which will only apply to tests in that folder.

In a test folder, define one or more fixtures in conftest.py, and they will be available to all tests in that folder, as well as any subfolders.

Hooks

Hooks are predefined functions that will automatically be called at various points in testing. All hooks start with *pytest_*. A pytest.Function object, which contains the actual test function, is passed into the hook.

For instance, `pytest_runtest_setup()` will be called before each test.

NOTE A complete list of hooks can be found here: https://docs.pytest.org/en/latest/reference.html#hooks

Plugins

There are many pytest plugins to provide helpers for testing code that uses common libraries, such as **Django** or **redis**.

You can register plugins in conftest.py like so:

```
pytest_plugins = "plugin1", "plugin2",
```

This will load the plugins.

Example

pytests/stuff/conftest.py

```
#!/usr/bin/env python
from pytest import fixture

@fixture
def common_fixture():  # user-defined fixture
    return "DATA"


def pytest_runtest_setup(item):  # predefined hook (all hooks start with __pytest__
    print("Hello from setup,", item)
```

Example

pytests/stuff/test_stuff.py

```
#!/usr/bin/env python
import pytest

def test_one():  # unit test that writes to STDOUT
    print("WHOOPEE")
    assert(1)

def test_two(common_fixture):   # unit test that uses fixture from conftest.py
    assert(common_fixture == "DATA")

if __name__ == '__main__':
    pytest.main([__file__, "-s"])   # run tests (without stdout/stderr capture) when this
script is run
```

pytests/stuff/test_stuff.py

```
============================ test session starts =============================
platform darwin -- Python 3.9.12, pytest-7.1.2, pluggy-1.0.0
PyQt5 5.15.7 -- Qt runtime 5.15.2 -- Qt compiled 5.15.2
rootdir: /Users/jstrick/curr/courses/python/common/examples/pytests, configfile:
pytest.ini
plugins: anyio-3.6.1, qt-4.1.0, remotedata-0.3.3, assert-utils-0.3.1, lambda-2.1.0,
astropy-header-0.2.1, fixture-order-0.1.4, common-subject-1.0.6, mock-3.8.2, typeguard-
2.13.3, astropy-0.10.0, filter-subpackage-0.1.1, hypothesis-6.54.3, openfiles-0.5.0,
django-4.5.2, doctestplus-0.12.0, cov-3.0.0, arraydiff-0.5.0
collected 2 items

pytests/stuff/test_stuff.py Hello from setup, <Function test_one>
WHOOPEE
.Hello from setup, <Function test_two>
.

============================== 2 passed in 0.03s ===============================
```

Parametrizing tests

- Run same test on multiple values
- Add parameters to fixture decorator
- Test run once for each parameter
- Use `pytest.mark.parametrize()`

Many tests require testing a method or function against many values. Rather than writing a loop in the test, you can automatically repeat the test for a set of inputs via **parametrizing**.

Apply the `@pytest.mark.parametrize` decorator to the test. The first argument is a string with the comma-separated names of the parameters; the second argument is the list of parameters. The test will be called once for each item in the parameter list. If a parameter list item is a tuple or other multi-value object, the items will be passed to the test based on the names in the first argument.

NOTE For more advanced needs, when you need some extra work to be done before the test, you can do indirect parametrizing, which uses a parametrized fixture. See `test_parametrize_indirect.py` for an example.

NOTE The authors of pytest deliberately spelled it "parametrizing", not "parameterizing".

Example

pytests/test_parametrization.py

```python
import pytest


def triple(x):  # Function to test
    return x * 3

test_data = [(5, 15), ('a', 'aaa'), ([True], [True, True, True])]  # List of values for testing containing input and expected result

@pytest.mark.parametrize("input,result", test_data)  # Parametrize the test with the test data; the first argument is a string defining parameters to the test and mapping them to the test data
def test_triple(input, result):  # The test expects two parameters (which come from each element of test data)
    print("input {} result {}:".format(input, result))  # The test expects two parameters (which come from each element of test data)
    assert triple(input) == result   # Test the function with the parameters


if __name__ == "__main__":
    pytest.main([__file__, '-s'])
```

pytests/test_parametrization.py

```
============================ test session starts =============================
platform darwin -- Python 3.9.12, pytest-7.1.2, pluggy-1.0.0
PyQt5 5.15.7 -- Qt runtime 5.15.2 -- Qt compiled 5.15.2
rootdir: /Users/jstrick/curr/courses/python/common/examples/pytests, configfile:
pytest.ini
plugins: anyio-3.6.1, qt-4.1.0, remotedata-0.3.3, assert-utils-0.3.1, lambda-2.1.0,
astropy-header-0.2.1, fixture-order-0.1.4, common-subject-1.0.6, mock-3.8.2, typeguard-
2.13.3, astropy-0.10.0, filter-subpackage-0.1.1, hypothesis-6.54.3, openfiles-0.5.0,
django-4.5.2, doctestplus-0.12.0, cov-3.0.0, arraydiff-0.5.0
collected 3 items

pytests/test_parametrization.py input 5 result 15:
.input a result aaa:
.input [True] result [True, True, True]:
.

============================= 3 passed in 0.02s ===============================
```

Marking tests

- Create groups of tests ("test cases")
- Can create multiple groups
- Use `@pytest.mark.somemark()`

You can mark tests with labels so that they can be run as a group. Use `@pytest.mark.marker()`, where *marker* is the marker (label), which can be any alphanumeric string.

Then you can run select tests which contain or match the marker, as described in the next topic.

In addition, you can register markers in the **[pytest]** section of **pytest.ini**, so they will be listed with `pytest --markers`:

```
[pytest]
markers =
  internet: test requires internet connection
  slow: tests that take more time (omit with '-m "not slow")
```

```
pytest -m "mark"
pytest -m "not mark"
```

Example

pytests/test_mark.py

```
import pytest

@pytest.mark.alpha  # Mark with label alpha
def test_one():
    assert 1

@pytest.mark.alpha  # Mark with label alpha
def test_two():
    assert 1

@pytest.mark.beta  # Mark with label beta
def test_three():
    assert 1

if __name__ == '__main__':
    pytest.main([__file__, '-m alpha'])  # Only tests marked with alpha will run
(equivalent to 'pytest -m alpha' on command line)
```

pytests/test_mark.py

```
============================= test session starts ==============================
platform darwin -- Python 3.9.12, pytest-7.1.2, pluggy-1.0.0
PyQt5 5.15.7 -- Qt runtime 5.15.2 -- Qt compiled 5.15.2
rootdir: /Users/jstrick/curr/courses/python/common/examples/pytests, configfile:
pytest.ini
plugins: anyio-3.6.1, qt-4.1.0, remotedata-0.3.3, assert-utils-0.3.1, lambda-2.1.0,
astropy-header-0.2.1, fixture-order-0.1.4, common-subject-1.0.6, mock-3.8.2, typeguard-
2.13.3, astropy-0.10.0, filter-subpackage-0.1.1, hypothesis-6.54.3, openfiles-0.5.0,
django-4.5.2, doctestplus-0.12.0, cov-3.0.0, arraydiff-0.5.0
collected 3 items / 1 deselected / 2 selected

pytests/test_mark.py ..                                                   [100%]

======================= 2 passed, 1 deselected in 0.02s ========================
```

Running tests (advanced)

- Run all tests
- Run by
  - function
  - class
  - module
  - name match
  - group

pytest provides many ways to select which tests to run.

Running all tests

To run all tests in the current and any descendent directories, use

Use **-s** to disable capturing, so anything written to STDOUT is displayed. Use **-s** for verbose output.

```
pytest
pytest -v
pytest -s
pytest -vs
```

Running by component

Use the node ID to select by component, such aas module, class, method, or function name:

```
file::class
file::class::test
file::::test
```

```
pytest test_president.py::test_dates
pytest test_president.py::test_dates::test_birth_date
```

Running by name match

Use **-k** to run all tests whose name includes a specified string

`pytest -k date` *run all tests whose name includes 'date'*

Skipping and failing

- Conditionally skip tests
- Completely ignore tests
- Decorate with
  - @pytest.mark.xfail
  - @pytest.mark.skip

To skip tests conditionally (or unconditionally), use `@pytest.mark.skip()`. This is useful if some tests rely on components that haven't been developed yet, or for tests that are platform-specific.

To fail on purpose, use `@pytest.mark.xfail)`. This reports the test as "XPASS" or "xfail", but does not provide traceback. Tests marked with xfail will not fail the test suite. This is useful for testing not-yet-implemented features, or for testing objects with known bugs that will be resolved later.

Example

pytests/test_skip.py

```
import sys
import pytest

def test_one():  # Normal test
    assert 1

@pytest.mark.skip(reason="can not currently test")  # Unconditionally skip this test
def test_two():
    assert 1

@pytest.mark.skipif(sys.platform != 'win32', reason="only implemented on Windows")  #
Skip this test if current platform is not Windows
def test_three():
    assert 1

@pytest.mark.xfail  ④
def test_four():
    assert 1

@pytest.mark.xfail  ④
def test_five():
    assert 0

if __name__ == '__main__':
    pytest.main([__file__, '-v'])
```

pytests/test_skip.py

```
============================= test session starts ==============================
platform darwin -- Python 3.9.12, pytest-7.1.2, pluggy-1.0.0 --
/Users/jstrick/opt/miniconda3/bin/python
cachedir: .pytest_cache
PyQt5 5.15.7 -- Qt runtime 5.15.2 -- Qt compiled 5.15.2
hypothesis profile 'default' ->
database=DirectoryBasedExampleDatabase('/Users/jstrick/curr/courses/python/common/example
s/.hypothesis/examples')
rootdir: /Users/jstrick/curr/courses/python/common/examples/pytests, configfile:
pytest.ini
plugins: anyio-3.6.1, qt-4.1.0, remotedata-0.3.3, assert-utils-0.3.1, lambda-2.1.0,
astropy-header-0.2.1, fixture-order-0.1.4, common-subject-1.0.6, mock-3.8.2, typeguard-
2.13.3, astropy-0.10.0, filter-subpackage-0.1.1, hypothesis-6.54.3, openfiles-0.5.0,
django-4.5.2, doctestplus-0.12.0, cov-3.0.0, arraydiff-0.5.0
collecting ... collected 5 items

pytests/test_skip.py::test_one PASSED                                    [ 20%]
pytests/test_skip.py::test_two SKIPPED (can not currently test)          [ 40%]
pytests/test_skip.py::test_three SKIPPED (only implemented on Windows)   [ 60%]
pytests/test_skip.py::test_four XPASS                                    [ 80%]
pytests/test_skip.py::test_five XFAIL                                    [100%]

============== 1 passed, 2 skipped, 1 xfailed, 1 xpassed in 0.17s ==============
```

Mocking data

- Simulate behavior of actual objects
- Replace expensive dependencies (time/resources)
- Use **unittest.mock** or **pytest-mock**

Some objects have dependencies which can make unit testing difficult. These dependencies may be expensive in terms of time or resources.

The solution is to use a **mock** object, which pretends to be the real object. A mock object behaves like the original object, but is restricted and controlled in its behavior.

For instance, a class may have a dependency on a database query. A mock object may accept the query, but always returns a hard-coded set of results.

A mock object can record the calls made to it, and assert that the calls were made with correct parameters.

A mock object can be preloaded with a return value, or a function that provides dynamic (or random) return values.

A *stub* is an object that returns minimal information, and is also useful in testing. However, a mock object is more elaborate, with record/playback capability, assertions, and other features.

pymock objects

- Use pytest-mock plugin
  - Can also use unittest.mock.Mock
- Emulate resources

pytest can use **unittest.mock**, from the standard library, or the **pytest-mock** plugin, which provides a wrapper around unittest.mock

Once the pytest-mock module is installed, it provides a fixture named **mocker**, from which you can create mock objects.

In either case, there are two primary ways of using mock. One is to provide a replacement class, function, or data object that mimics the real thing.

The second is to monkey-patch a library, which temporarily (just during the test) replaces a component with a mock version. The **mocker.patch()** function replaces a component with a mock object. Any calls to the component are now recorded.

Example

pytests/test_mock_unittest.py

```python
import pytest
from subjectlib import Spam
import hamlib

HAM_VALUE = 42

def test_spam_calls_ham(mocker):
    mocker.patch('hamlib.ham', return_value=HAM_VALUE * 10)
    s = Spam(HAM_VALUE)  # Create instance of Spam, which calls ham()
    assert s.value == HAM_VALUE * 10

if __name__ == '__main__':
    pytest.main([__file__, '-s'])   # Start the test runner
```

pytests/test_mock_unittest.py

```
============================= test session starts ==============================
platform darwin -- Python 3.9.12, pytest-7.1.2, pluggy-1.0.0
PyQt5 5.15.7 -- Qt runtime 5.15.2 -- Qt compiled 5.15.2
rootdir: /Users/jstrick/curr/courses/python/common/examples/pytests, configfile:
pytest.ini
plugins: anyio-3.6.1, qt-4.1.0, remotedata-0.3.3, assert-utils-0.3.1, lambda-2.1.0,
astropy-header-0.2.1, fixture-order-0.1.4, common-subject-1.0.6, mock-3.8.2, typeguard-
2.13.3, astropy-0.10.0, filter-subpackage-0.1.1, hypothesis-6.54.3, openfiles-0.5.0,
django-4.5.2, doctestplus-0.12.0, cov-3.0.0, arraydiff-0.5.0
collected 1 item

pytests/test_mock_unittest.py .

============================== 1 passed in 0.02s ===============================
```

Example

pytests/test_mock_pymock.py

```
import pytest  # Needed for test runner
import re  # Needed for test (but will be mocked)
from subjectlib import SpamSearch   # subject under test


def test_spam_search_calls_re_search(mocker):   # Unit test
    mocker.patch('re.search')  # Patch re.search (i.e., replace re.search with a Mock
object that records calls to it)
    s = SpamSearch('bug', 'lightning bug')  # Create instance of SpamSearch
    _ = s.findit()   # Call the method under test
    re.search.assert_called_once_with('bug', 'lightning bug')  # Check that method was
called just once with the expected parameters

if __name__ == '__main__':
    pytest.main([__file__, '-s'])   # Start the test runner
```

pytests/test_mock_pymock.py

```
============================= test session starts ==============================
platform darwin -- Python 3.9.12, pytest-7.1.2, pluggy-1.0.0
PyQt5 5.15.7 -- Qt runtime 5.15.2 -- Qt compiled 5.15.2
rootdir: /Users/jstrick/curr/courses/python/common/examples/pytests, configfile:
pytest.ini
plugins: anyio-3.6.1, qt-4.1.0, remotedata-0.3.3, assert-utils-0.3.1, lambda-2.1.0,
astropy-header-0.2.1, fixture-order-0.1.4, common-subject-1.0.6, mock-3.8.2, typeguard-
2.13.3, astropy-0.10.0, filter-subpackage-0.1.1, hypothesis-6.54.3, openfiles-0.5.0,
django-4.5.2, doctestplus-0.12.0, cov-3.0.0, arraydiff-0.5.0
collected 1 item

pytests/test_mock_pymock.py .

============================== 1 passed in 0.02s ===============================
```

Example

pytests/test_mock_play.py

```python
import pytest
from unittest.mock import Mock


@pytest.fixture
def small_list():   # Create fixture that provides a small list
    return [1, 2, 3]


def test_m1_returns_correct_list(small_list):
    m1 = Mock(return_value=small_list)  # Create mock object that "returns" a small list
    mock_result = m1('a', 'b')  # Call mock object with arbitrary parameters
    assert mock_result == small_list  # Check the mocked result


m2 = Mock()  # Create generic mock object

m2.spam('a', 'b')  # Call fake methods on mock object
m2.ham('wombat')  # Call fake methods on mock object
m2.eggs(1, 2, 3)  # Call fake methods on mock object

print("mock calls:", m2.mock_calls)  # Mock object remembers all calls

m2.spam.assert_called_with('a', 'b')  # Assert that spam() was called with parameters 'a'
and 'b'
```

pytests/test_mock_play.py

```
mock calls: [call.spam('a', 'b'), call.ham('wombat'), call.eggs(1, 2, 3)]
```

Pytest plugins

- Common plugins
  - **pytest-qt**
  - **pytest-django**

There are some plugins for **pytest** that that integrate various frameworks which would otherwise be difficult to test directly.

The **pytest-qt** plugin provides a `qtbot` fixture that can attach widgets and invoke events. This makes it simpler to test your custom widgets.

The **pytest-django** plugin allows you to run Django with **pytest**-style tests rather than the default **unittest** style.

See https://docs.pytest.org/en/latest/reference/plugin_list.html for a complete list of plugins. There are currently 880 plugins!

Pytest and Unittest

- Run Unittest-based tests
- Use Pytest test runner

The Pytest builtin test runner will detect Unittest-based tests as well. This can be handy for transitioning legacy code to Pytest.

Chapter 7 Exercises

Exercise 7-1 (test_president_pytest.py)

Using **pytest**, Create some unit tests for the President class you created earlier.[1]

Suggestions for tests:

- What happens when an out-of-range term number is given?
- President 1’s first name is "George"
- All presidential terms match the correct last name (use list of last names and **parametrize**)
- Confirm date fields return an object of type **datetime.date**

[1] If there was not an exercise where you created a President class, you can use **president.py** in the top-level folder of the student guide.

Chapter 8: Design Patterns

Objectives

- Examine important OOP principles
- Understand why design patterns are useful
- Learn common design patterns
- Implement design patterns in Python

Coupling and cohesion

- Coupling: interdependence of two components
- Cohesion: how well components fit together
- Strive for looser coupling but tighter cohesion

Coupling and **cohesion** are two terms often used when studying design patterns. They refer to the way in which components interact.

Coupling describes how much (or how little) two components depend on each other. The more dependence, the greater the complexity, and the more that one has to change when the other changes. Components are tightly coupled if they share variables, or exchange information that alters their behavior.

Thus, one primary design goal for software components is looser coupling. This means that components can interact, but are independent of each other. An example of this is illustrated by the Strategy pattern.

Cohesion describes how well components fit together. In the ideal world, each component should implement just one function or provide just one data value. The more responsibilities a component has, the less cohesive it is.

Interfaces

- Do not exist as such in Python
- Also known as abstract classes (but not in Java)
- Can be implemented with the abc module

An important software design rule is to program to the **interface**, not the object.

Code which expects a particular object type is tightly coupled to that type, and any changes to the object type will require changes to the code.

For example, rather than building a logger that writes to a text file, it is better to build a logger that writes to an object that acts like, but is not necessarily, a text file. In some languages, we would say that the logger uses the text file interface, but Python does not directly support interfaces.

Being a dynamic language, Python uses "duck" typing, which means that any object providing the appropriate methods can be substituted for the expected object.

Thus, our logger can write to any object that implements a text-file-like interface, in the informal sense.

This is the Pythonic way, and works well, especially if components are well documented.

If you want to ensure that a class implements a particular set of methods, you can use the **abc** module. It allows you to create abstract base classes. These classes may not be directly instantiated, and if abstract methods are not overwritten, an exception is raised upon instantiation.

Whenever you want to enforce that "interface", add the abstract class to the base class list. This is possible because Python supports multiple inheritance.

Example

abstract_base_classes.py

```
from abc import ABCMeta, abstractmethod

class Animal(metaclass=ABCMeta):  # metaclasses control how classes are created; ABCMeta
adds restrictions to classes that inherit from Animal

    @abstractmethod   # when decorated with @abstractmethod, speak() becomes an abstract
method
    def speak(self):
        pass

class Dog(Animal):  # Inherit from abstract base class Animal
    def speak(self):   # speak() must be implemented
        print("woof! woof!")

class Cat(Animal):  # Inherit from abstract base class Animal
    def speak(self):  # speak() must be implemented
        print("Meow meow meow")

class Duck(Animal): # Inherit from abstract base class Animal
    pass  # Duck does not implement speak()

d = Dog()
d.speak()

c = Cat()
c.speak()

try:
    d = Duck()  # Duck throws a TypeError if instantiated
    d.speak()
except TypeError as err:
    print(err)
```

abstract_base_classes.py

```
woof! woof!
Meow meow meow
Can't instantiate abstract class Duck with abstract method speak
```

What are design patterns?

- Reusable solution pattern
- Not a design or specification
- Not an implementation
- A description of how to solve a problem
- A formalized best practice

A **design pattern** is basically a "way of doing something". Design patterns grow from observing the best ways to solve problems.

A design pattern is not a specification for a particular software node; it is a description of how to do something. It is more than an algorithm, because algorithms only describe logic. Design patterns describe the big-picture details of how to implement software for common use cases.

Design patterns became well-known in the computer science world after the publication of the book Design Patterns: Elements of Reusable Object-Oriented Software, published in 1994. The four authors are typically referred to as the "Gang of Four", and the book itself as the "GoF".

Design patterns are associated with object-oriented software, but are not restricted to OOP. The GoF book is pretty dry reading, and the examples are in Smalltalk and C++. For a newer take on design patterns, the following books are excellent:

- Mastering Python Design patterns
- Learning Python Design Patterns
- Head First Design Patterns (book examples are in Java)

NOTE find details for these books in Appendix A – Bibliography

Why use design patterns?

- Tried and true
- Vocabulary for discussing solutions
- Provide structure
- Higher-level than packages of actual code

Design patterns provide workable approaches to common coding problems. They help developers avoid mistakes, because the patterns have been discovered and refined by the actual experience of other programmers.

Another advantage is that design patterns provide a vocabulary for discussing the architecture of a group of related modules (e.g., classes or packages). One programmer can say "I think an Observer would work great for publishing events from our stock market widget", and another programmers will know what that means. It becomes a shorthand for standard approaches.

Remember that design patterns are design patterns. They operate at a higher level than packages and frameworks, which frequently contain specific implementations of patterns.

A real bonus when programming in Python is that many design patterns are already built into the language. Examples of these are Singletons (AKA modules) and Decorators.

Types of patterns

- Four categories of patterns (or so)
    - Creational
    - Structural
    - Behavioral
    - Concurrency

There are four general categories of patterns, organized by how they are used within a project.

Creational patterns are used for software that creates objects and other entities. Well-known creational patterns include Abstract Factory and Singleton.

Structural patterns are used to create relationships between software components. Common structural patterns are Adapter, Proxy, and Flyweight.

Behavioral patterns are used to interact between components. Common behavioral patterns are Observer, Command, and Visitor.

Concurrency patterns are used to help coordinate between components used in multiprogramming. Examples are Double-checked locking, Object Pool, and Active object.

These are not the only ways to organize design patterns. Others have created more categories, or organized them in a totally different hierarchy.

Table 7. Creational Patterns

Pattern	Description
Abstract factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Builder	Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.
Factory method	Define an interface for creating a single object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses (dependency injection[15]).
Lazy initialization	Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. This pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the Proxy pattern.
Multiton	Ensure a class has only named instances, and provide a global point of access to them.
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalization of connection pool and thread pool patterns.
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
Resource acquisition	Ensure that resources are properly released by tying them to the lifespan of suitable objects.
Singleton	Ensure a class has only one instance, and provide a global point of access to it

Table 8. Structural Patterns

Pattern	**Description**
Adapter or Wrapper or Translator	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.
Bridge	Decouple an abstraction from its implementation allowing the two to vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to sub-classing for extending functionality.
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Flyweight	Use sharing to support large numbers of similar objects efficiently.
Front Controller	The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.
Module	Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity.
Proxy	Provide a surrogate or placeholder for another object to control access to it.
Twin	Twin allows modeling of multiple inheritance in programming languages that do not support this feature.

Table 9. Behavioral Patterns

Pattern	Description
Blackboard	Artificial intelligence pattern for combining disparate sources of data (see blackboard system)
Chain of responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
Memento	Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.
Null object	Avoid null references by providing a default object.
Observer or Publish/subscribe	Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.
Servant	Define common functionality for a group of classes.
Specification	Recombinable business logic in a Boolean fashion.
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Template method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Table 10. Concurrency Patterns

Pattern	Description
Active Object	Decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.
Balking	Only execute an action on an object when the object is in a particular state.
Binding properties	Combining multiple observers to force properties in different objects to be synchronized or coordinated in some way.[19]
Block chain	Decentralized way to store data and agree on ways of processing it in a Merkle tree, optionally using Digital signature for any individual contributions.
Double-checked locking	Reduce the overhead of acquiring a lock by first testing the locking criterion (the 'lock hint') in an unsafe manner; only if that succeeds does the actual locking logic proceed.
Event-based asynchronous	Addresses problems with the asynchronous pattern that occur in multi-threaded programs.[20]
Guarded suspension	Manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.
Join	Join-pattern provides a way to write concurrent, parallel and distributed programs by message passing. Compared to the use of threads and locks, this is a high level programming model.
Lock	One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it.[21]
Messaging design pattern (MDP)	Allows the interchange of information (i.e. messages) between components and applications.
Monitor object	An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time.
Reactor	A reactor object provides an asynchronous interface to resources that must be handled synchronously.
Read-write lock	Allows concurrent read access to an object, but requires exclusive access for write operations.
Scheduler	Explicitly control when threads may execute single-threaded code.
Thread pool	A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Can be considered a special case of the object pool pattern.
Thread-specific storage	Static or "global" memory local to a thread.

Singleton

- Ensures only one instance is created
- Singleton class requires metaprogramming in Python
- Variation called the Borg
- A Python module is already a singleton

The Singleton pattern is used to provide global access to a resource, when it is expensive, or incorrect, to have multiple copies available. The singleton can also control concurrent access.

Typical use cases for the singleton are loggers, spoolers, etc., as well as database or network connections.

There are three standard ways to implement Single classes in Python: using a module (no classes involved), a "classic" singleton, or a Borg.

Module as Singleton

- Modules are only loaded once
- Global variables in module are available after import

Since a python module is already, by design, a singleton. Rather than creating a Singleton class, you can just put data and methods in a normal module, and the module can be imported from anywhere in the application. The data and methods will only exist in one place, and there will never be more than one instance. This is useful for simple cases, but it is limited in some ways.

Example

designpatterns/singletons/singletonmodule/dbconn.py

```python
#!/usr/bin/env python
# (c)2015 John Strickler

# only one connection and one cursor will be created

import sqlite3

db_connection = sqlite3.connect(':memory:')

db_cursor = db_connection.cursor()
```

Example

designpatterns/singletons/singletonmodule/singleton_main.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

from dbconn import db_cursor
from builddb import build_database

build_database()

db_cursor.execute(
    '''
        select first_name, last_name
        from computer_people
    '''
)

for row in db_cursor.fetchall():
    print(' '.join(row))
```

designpatterns/singletons/singletonmodule/singleton_main.py

```
Bill Gates
Steve Jobs
Paul Allen
Larry Ellison
Mark Zuckerberg
Sergey Brin
Larry Page
Linux Torvalds
```

Classic Singleton

- Override *new*
- *init* is still called for each instance

To create a classic singleton, override the *new* method of a class. This method is normally not used in creating classes, because the *init*() takes the role of constructor, but *new*() is the "real" constructor, in that it returns the actual new object when an instance is requested.

One thing to be careful about is that *init*() is still called for each instance, so put a test in *init*() to make sure you're only initializing the instance once.

Example

designpatterns/singletons/singletonclassic/databaseconnection.py

```python
#!/usr/bin/env python
# (c)2015 John Strickler

# only one connection and one cursor will be created

import sqlite3

class DatabaseConnection(object):
    def __new__(cls):
        if not hasattr(cls, 'instance'):
            cls.instance = super(DatabaseConnection, cls).__new__(cls)
        return cls.instance

    def __init__(self):
        # IMPORTANT! One-time setup
        if not hasattr(self, '_db_connection'):
            self._db_connection = sqlite3.connect(':memory:')
            self._db_cursor = self._db_connection.cursor()

    @property
    def cursor(self):
        return self._db_cursor
```

Example

designpatterns/singletons/singletonclassic/builddb.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

# even though we are importing DatabaseConnection, it does not
# create a new connection
from databaseconnection import DatabaseConnection

db_conn = DatabaseConnection()  # get the instance

PEOPLE = [
    ('Bill', 'Gates', 'Microsoft'),
    ('Steve', 'Jobs', 'Apple'),
    ('Paul', 'Allen', 'Microsoft'),
    ('Larry', 'Ellison', 'Oracle'),
    ('Mark', 'Zuckerberg', 'Facebook'),
    ('Sergey','Brin', 'Google'),
    ('Larry', 'Page', 'Google'),
    ('Linux', 'Torvalds', 'Linux'),
]

def build_database():
    db_conn.cursor.execute('''
        create table computer_people (
            first_name varchar(16),
            last_name varchar(16),
            known_for varchar(16)
        );
    ''')

    insert_query = '''
        insert into computer_people
        (first_name, last_name, known_for)
        values (?, ?, ?);
    '''

    db_conn.cursor.executemany(insert_query, PEOPLE)
```

Example

designpatterns/singletons/singletonclassic/singleton_main.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

from databaseconnection import DatabaseConnection

from builddb import build_database

build_database()

db_conn = DatabaseConnection()

db_conn.cursor.execute(
    '''
        select first_name, last_name
        from computer_people
    '''
)

for row in db_conn.cursor.fetchall():
    print(' '.join(row))
```

designpatterns/singletons/singletonclassic/singleton_main.py

```
Bill Gates
Steve Jobs
Paul Allen
Larry Ellison
Mark Zuckerberg
Sergey Brin
Larry Page
Linux Torvalds
```

The Borg

- Singleton that creates multiple instances
- BUT, each instance shares same state with all others
- Allows inheritance from singleton
- AKA monostate

If you don’t need to inherit from a Singleton, then the classic implementation is good. However, when you need to inherit, you can use the Borg implementation, named for the alien race on Star Trek TNG.

In the Borg, each instance is a distinct object, but all of the instances share the same state.

This is done by sharing data dictionaries at the class level. The *new*() method returns a new instance object, but replaces the default dictionary with the shared one. Thus, all Borg instances really share the same set of attributes.

Example

designpatterns/singletons/singletonborg/databaseconnection.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

import sqlite3

class DatabaseConnection(object):
    _shared = {}

    def __new__(cls, *args, **kwargs):
        instance = super(DatabaseConnection, cls).__new__(cls, *args, **kwargs)

        if not '_db_connection' in cls._shared:
            db_conn = sqlite3.connect(':memory:')
            db_cursor = db_conn.cursor()

            cls._shared['_db_connection'] = db_conn
            cls._shared['_db_cursor'] = db_cursor

        instance.__dict__ = cls._shared


        return instance

    @property
    def cursor(self):
        return self._db_cursor
```

Example

designpatterns/singletons/singletonborg/builddb.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

# even though we are importing DatabaseConnection, it does not
# create a new connection
from databaseconnection import DatabaseConnection

db_conn = DatabaseConnection()  # get the instance

PEOPLE = [
    ('Bill', 'Gates', 'Microsoft'),
    ('Steve', 'Jobs', 'Apple'),
    ('Paul', 'Allen', 'Microsoft'),
    ('Larry', 'Ellison', 'Oracle'),
    ('Mark', 'Zuckerberg', 'Facebook'),
    ('Sergey','Brin', 'Google'),
    ('Larry', 'Page', 'Google'),
    ('Linux', 'Torvalds', 'Linux'),
]

def build_database():
    db_conn.cursor.execute('''
        create table computer_people (
            first_name varchar(16),
            last_name varchar(16),
            known_for varchar(16)
        );
    ''')

    insert_query = '''
        insert into computer_people
        (first_name, last_name, known_for)
        values (?, ?, ?);
    '''

    db_conn.cursor.executemany(insert_query, PEOPLE)
```

Example

designpatterns/singletons/singletonborg/singleton_main.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

from databaseconnection import DatabaseConnection

from builddb import build_database

build_database()

db_conn = DatabaseConnection()

db_conn.cursor.execute(
    '''
        select first_name, last_name
        from computer_people
    '''
)

for row in db_conn.cursor.fetchall():
    print(' '.join(row))
```

designpatterns/singletons/singletonborg/singleton_main.py

```
Bill Gates
Steve Jobs
Paul Allen
Larry Ellison
Mark Zuckerberg
Sergey Brin
Larry Page
Linux Torvalds
```

Strategy

- Encapsulate algorithms to make them interchangeable
- Isolate the things that vary from the things that don't
- Provides an interface for behavior
- Encourages loose coupling

The Strategy pattern is used when you have behaviors that may vary, but the use of the behavior stays the same in a client.

The client may want to sort, but we may want to be able to change the sort algorithm independently, from quicksort to mergesort to Timsort. Strategy decouples the sorting algorithm from the component that wants to sort.

The client gets and instance of the algorithm component, which is typically a class written to an API (an interface, in some languages). As long as the class provides the correct API, the client doesn't care about the algorithm's details.

Another example is saving a file from a word processor. The interface can select a class that saves in native format, XML, PDF, plain text, or translates to Klingon. All it has to do is get an instance of NativeWriter, XMLWriter, PDFWriter, PlainTextWriter, or KlingonWriter. No matter what kind of writer, it just passes the current document to the writer's write() method. Of course in this case the writer component will need to be able to understand the in-memory document format.

Example

designpatterns/strategies/simuduck/flylib.py

```python
#!/usr/bin/env python
# (c)2015 John Strickler

from abc import ABCMeta, abstractmethod

class FlyBase(object, metaclass=ABCMeta):
    @abstractmethod
    def fly(self):
        pass

class FlyWithWings(FlyBase):
    def fly(self):
        print("Soaring on my wings")

class Flightless(FlyBase):
    def fly(self):
        print("Gee, I can't fly")

if __name__ == '__main__':
    fww = FlyWithWings()
    fww.fly()

    fl = Flightless()
    fl.fly()
```

Example

designpatterns/strategies/simuduck/quacklib.py

```python
#!/usr/bin/env python
# (c)2015 John Strickler
from abc import ABCMeta, abstractmethod

class QuackBase(object, metaclass=ABCMeta):
    @abstractmethod
    def quack(self):
        pass

class Quack(QuackBase):
    def quack(self):
        print('"Quack, quack"')

class Squeak(QuackBase):
    def quack(self):
        print('"Squeaky-squeaky"')

class MuteQuack(QuackBase):
    def quack(self):
        print("<I can't make any sounds>")

if __name__ == '__main__':
    q = Quack()
    q.quack()

    s = Squeak()
    s.quack()

    m = MuteQuack()
    m.quack()
```

Example

designpatterns/strategies/simuduck/duck.py

```
#!/usr/bin/env python
# (c)2015 John Strickler
import flylib
import quacklib

class Duck(object):

    def __init__(
            self,
            duck_type,
            quack=None,
            fly=None
    ):
        self._duck_type = duck_type
        self._quack_behavior = quacklib.Quack() if quack is None else quack
        self._fly_behavior = flylib.FlyWithWings() if fly is None else fly


    def swim(self):
        print("Swimming")

    def display(self):
        print("I am a", self._duck_type)

    def set_quack(self, quack):
        if not issubclass(quack, quacklib.QuackBase):
            raise TypeError("Invalid quack")
        self._quack_behavior = quack

    def set_fly(self, fly):
        if not issubclass(fly, flylib.FlyBase):
            raise TypeError("Invalid flight type")
        self._fly_behavior = fly

    def quack(self):
        self._quack_behavior.quack()

    def fly(self):
        self._fly_behavior.fly()
```

Example

designpatterns/strategies/simuduck/simuduck_main.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

from duck import Duck
from quacklib import MuteQuack, Squeak
from flylib import Flightless

d1 = Duck('Mallard')
d2 = Duck('Robot', quack=MuteQuack())
d3 = Duck('Rubber Duckie', quack=Squeak(), fly=Flightless())

for d in d1, d2, d3:
    d.display()
    d.quack()
    d.fly()
    print('-' * 60)
```

designpatterns/strategies/simuduck/simuduck_main.py

```
I am a Mallard
"Quack, quack"
Soaring on my wings
------------------------------------------------------------
I am a Robot
<I can't make any sounds>
Soaring on my wings
------------------------------------------------------------
I am a Rubber Duckie
"Squeaky-squeaky"
Gee, I can't fly
------------------------------------------------------------
```

Decorators

- Built into Python
- Implemented via functions or classes
- Can decorate functions or classes
- Can take parameters (but not required to)
- functools.wraps() preserves function’s properties

In Python, decorators are a special case. Not only are decorators provided by the standard library, such as property() or classmethod(), but they have a special syntax. The @ sign is used to apply a decorator to a function or class.

A decorator is a component that modifies some other component. The purpose is typically to add functionality, but there are no real restrictions on what a decorator can do. Many decorators register a component with some other component. For instance, the @app.route() decorator in Flask maps a URL to a view function.

Table 11. Decorators in the standard library

Decorator	Description
`@abc.abstractmethod`	Indicate abstract method (must be implemented).
`@abc.abstractproperty`	Indicate abstract property (must be implemented). **DEPRECATED**
`@asyncio.coroutine`	Mark generator-based coroutine.
`@atexit.register`	Register function to be executed when interpreter (script) exits.
`@classmethod`	Indicate class method (receives class object, not instance object)
`@contextlib.contextmanager`	Define factory function for **with** statement context managers (no need to create __enter__() and __exit__() methods)
`@functools.lru_cache`	Wrap a function with a memoizing callable
`@functools.singledispatch`	Transform function into a single-dispatch generic function.
`@functools.total_ordering`	Supply all other comparison methods if class defines at least one.
`@functools.wraps`	Invoke update_wrapper() so decorator's replacement function keeps original function's name and other properties.
`@property`	Indicate a class property.
`@staticmethod`	Indicate static method (passed neither instance nor class object).
`@types.coroutine`	Transform generator function into a coroutine function.
`@unittest.mock.patch`	Patch target with a new object. When the function/with statement exits patch is undone.
`@unittest.mock.patch.dict`	Patch dictionary (or dictionary-like object), then restore to original state after test.
`@unittest.mock.patch.multiple`	Perform multiple patches in one call.
`@unittest.mock.patch.object`	Patch object attribute with mock object.
`@unittest.skip()`	Skip test unconditionally
`@unittest.skipIf()`	Skip test if condition is true
`@unittest.skipUnless()`	Skip test unless condition is true
`@unittest.expectedFailure()`	Mark Test as expected failure
`@unittest.removeHandler()`	Remove Control-C handler

Using decorators

- Provide a wrapper around a function or class
- Syntax

```
@decorator
def function():
    function_body
```

A decorator is a function or class that acts as a wrapper around a function or class. It allows you to modify the target without changing the target itself.

The @property, @classmethod, and @staticmethod decorators were described in the chapter on OOP.

Table 12. Decorator implementations

Implemented as (wrapper)	Decorates (target)	Takes params	Implementation [1]	Example as function call (without @)
function	function	No	Decorator returns replacement	`target = decorator(target)`
function	function	Yes	Decorator returns wrapper function Wrapper returns replacement function	`target = decorator(params)(target)`
class	function	No	`__call__` ***IS*** replacement function	`target = decorator(target)`
class	function	Yes	`__call__` ***RETURNS*** replacement function	`target = decorator(params)(target)`
function	class	No	Decorator modifies and returns original class	`target = decorator(target)`
function	class	Yes	Decorator returns wrapper Wrapper modifies and returns original class	`target = decorator(params)(target)`
class	class	No	`__new__` returns original class	`target = decorator(target)`
class	class	Yes	`__call__` returns original class	`target = decorator(params)(target)`

[1]In all cases, target is decorated with @decorator or @decorator(params). The example in column 5 is equivalent to the @ syntax, but is not normally used.

NOTE See the file **EXAMPLES/decorators/decorama.py** for examples of the above 8 different decorator implementations.

Creating decorator functions

- Decorator function gets original function
- Use functools.wraps

A decorator function is passed the target object (function or class), and returns a replacement object.

A simple decorator function expects only one argument – the function to be modified. It should return a new function, which will replace the original. The replacement function typically calls the original function as well as some new code.

A decorated function, like

```
@mydecorator
def myfunction():
    pass
```

is really called like this

```
myfunction = mydecorator(myfunction)
```

The new function should be defined with generic arguments so it can handle the original function's arguments.

The replacement function should be decorated with functools wraps. This makes sure the replacement function keeps the name (and other attributes) of the original function.

Example

designpatterns/decorators/deco_print_name.py

```
#!/usr/bin/env python

from functools import wraps

def print_name( old_func ):

    @wraps(old_func)
    def new_func( *args, **kwargs ):
        # added functionality
        print("==> Calling function {0}".format(old_func.__name__))
        return old_func( *args, **kwargs )  # call the 'real' function

    return new_func   # return the new function object


@print_name
def hello():
    print("Hello!")

@print_name
def goodbye():
    print("Goodbye!")

hello()
goodbye()
hello()
goodbye()
```

designpatterns/decorators/deco_print_name.py

```
==> Calling function hello
Hello!
==> Calling function goodbye
Goodbye!
==> Calling function hello
Hello!
==> Calling function goodbye
Goodbye!
```

Decorators with arguments

If the decorator itself needs arguments, then things get a little bit trickier. The decorator function gets the arguments, and returns a wrapper function that gets the original function (the one being decorated), and the wrapper function then returns the replacement function.

That is,

```
@mydecorator(param)
def myfunction():
    pass
```

is really called like this

```
myfunction = mydecorator(param)(myfunction)
```

Example

designpatterns/decorators/deco_print_name_with_label.py

```
#!/usr/bin/env python

from functools import wraps

def print_name(label):

    def wrapper(old_func):

        @wraps(old_func)
        def new_func( *args, **kwargs ):
            # added functionality
            print("{0}: function {1}".format(
                label,
                old_func.__name__
            ))
            return old_func( *args, **kwargs )  # call the 'real' function

        return new_func   # return the new function object
    return wrapper

@print_name('HELLO')
def hello():
    print("Hello!")

@print_name('HELLO')
def howdy():
    print("Howdy!")

@print_name('GOODBYE')
def goodbye():
    print("Goodbye!")

@print_name('GOODBYE')
def solong():
    print("So long!")


hello()
howdy()
goodbye()
solong()
```

designpatterns/decorators/deco_print_name_with_label.py

```
HELLO: function hello
Hello!
HELLO: function howdy
Howdy!
GOODBYE: function goodbye
Goodbye!
GOODBYE: function solong
So long!
```

Adapters

- Help make two incompatible interfaces compatible
- Prevent making changes to either interface
- Work with objects after implementation
- Adapt one interface to the other

The Adapter pattern is used to make two incompatible interfaces compatible. This is needed when you are not able to change the software on one side or the other. For instance, you're using a third-party library that you can't change. Your software was written for an older version of the library, and when they update the library, suddenly your code is broken. While in the ideal world, you could update the actual component that needs the library, an Adapter can let your existing code work with the updated library.

It can also be used for data conversions. Let's say you're getting data from a web service, but the units are kilometers and you need them in miles. You can create an adapter that connects to the original service, and converts the data before returning it to the requesting component.

An Adapter takes an instance of the caller, and uses the information in the instance to make the appropriate calls to the callee.

In the Java library, the IO readers and writers are adapters.

A Proxy is similar to an Adapter, but the Proxy presents the same interface as the component it represents, while an Adapter does not.

Example

designpatterns/adapters/cat_adapter.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

class Cat(object):
    def meow(self):
        print("Meow!!")

class Dog(object):
    def bark(self):
        print("Arf arf!!")

class CatAdapter(Dog):
    def __init__(self, cat):
        self._cat = cat

    def bark(self):
        self._cat.meow()

Garfield = Cat()

dog = CatAdapter(Garfield)
dog.bark()
```

designpatterns/adapters/cat_adapter.py

```
Meow!!
```

Abstract Factory

- Create families of objects
- Meta-factory (factory of factories)
- Provide various objects that share API
- Alias: Kit
- Use cases: – System should be independent of how products are created – System configured with one of multiple families of products – Enforce constraints on family of related products – Create abstract classes for product family

Participating objects: Abstract Factory, Factory, Abstract Product, Product, Client

An Abstract Factory is a creational pattern that is useful when you want to decouple the creation of classes from an application (or other component).

An Abstract Factory can be considered a meta-factory in that it typically returns a specific factory which creates the desired object. The client code passes a parameter to the Abstract Factory to tell it which factory to return.

While a Factory uses inheritance, an Abstract Factory uses composition.

The official GoF definition of the Abstract Factory is:

“Provide an interface for creating families of related or dependent objects without specifying their concrete classes.”

Builder

- Separate construction from representation
- Creates object composed of multiple parts
- Object not complete until all parts created
- Same builder can create multiple representations
- Use cases
    - Product creation algorithm should be independent of parts and assembly
    - Construction process should allow different representations of product

Participating objects: Builder Mixin, Builder, Director, Product

A Builder is a creational pattern that separates the construction of a product from its representation. A builder object creates the multiple parts of the target object. A director object controls the building of an object by using an instance of a builder object.

The representation of the product can vary, but the process remains the same.

Example

builder_pattern.py

```
from abc import ABCMeta, abstractmethod


class Director(object):

    def construct(self, builder):
        '''
        Builder uses multiple steps
        :param builder: A Builder object
        :return:
        '''
        builder.build_part_A()
        builder.build_part_B()

class BuilderBase(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def build_part_A(self): pass

    @abstractmethod
    def build_part_B(self): pass

    @abstractmethod
    def get_result(self): pass

class Product(object):
    def __init__(self, name):
        self._name = name
        self._parts = []

    def add(self, part):
        self._parts.append(part)

    def show(self):
        print("{} Parts -------".format(self._name))
        for part in self._parts:
            print(part)
        print()

class Builder1(BuilderBase):
    def __init__(self):
        self._product = Product("Product 1")
    def build_part_A(self):
        self._product.add("PartA")

    def build_part_B(self):
        self._product.add("PartB")

    def get_result(self):
        return self._product


class Builder2(BuilderBase):
    def __init__(self):
        self._product = Product("Product 2")

    def build_part_A(self):
        self._product.add("PartX")

    def build_part_B(self):
        self._product.add("PartY")

    def get_result(self):
        return self._product

if __name__ == '__main__':
    director = Director()
    builder1 = Builder1()
    builder2 = Builder2()

    director.construct(builder1)
    product1 = builder1.get_result()
    product1.show()

    director.construct(builder2)
    product2 = builder2.get_result()
    product2.show()
```

builder_pattern.py

```
Product 1 Parts -------
PartA
PartB

Product 2 Parts -------
PartX
PartY
```

Facade

- Simplified front end to a system
- Hides the details of a complex component
- Implemented as class or function
- Use cases – Simple interface for complex subsystem – Decouple subsystem from other subsystems – Create layered subsystems

Participating objects: Facade, subsystem classes

A Facade is a design pattern that creates a simplified interface for a complex component. It essentially hides the details of a system.

For instance, the requests module is a Facade for the urllib.* packages. requests makes proxies, authentication, and sending data much easier than doing everything individually.

Another name for a Facade could be "wrapper". It "wraps" some other component. One downside to a Facade is that it might not provide access to all the features available in the target component. A Facade provides a new interface to the component.

Example

csv_facade.py

```
#
"""
Demo of the Facade design pattern
"""
import csv

PRESIDENTS_FILE = '../DATA/presidents.csv'

class CSVUpper():
    """
    A facade for a CSV file
    """
    def __init__(self, csv_file):
        self._file_in = open(csv_file)
        self._rdr = csv.reader(self._file_in)

    def __iter__(self):
        return self

    def __next__(self):
        row = next(self._rdr)
        if row:
            return [r.upper() for r in row]
        else:
            raise StopIteration()

    def __del__(self):
        self._file_in.close()

if __name__ == '__main__':
    presidents = CSVUpper(PRESIDENTS_FILE)
    for p in presidents:
        print(p)
```

csv_facade.py

```
['1', 'GEORGE', 'WASHINGTON', 'WESTMORELAND COUNTY', 'VIRGINIA', 'NO PARTY']
['2', 'JOHN', 'ADAMS', 'BRAINTREE, NORFOLK', 'MASSACHUSETTS', 'FEDERALIST']
['3', 'THOMAS', 'JEFFERSON', 'ALBERMARLE COUNTY', 'VIRGINIA', 'DEMOCRATIC - REPUBLICAN']
['4', 'JAMES', 'MADISON', 'PORT CONWAY', 'VIRGINIA', 'DEMOCRATIC - REPUBLICAN']
['5', 'JAMES', 'MONROE', 'WESTMORELAND COUNTY', 'VIRGINIA', 'DEMOCRATIC - REPUBLICAN']
['6', 'JOHN QUINCY', 'ADAMS', 'BRAINTREE, NORFOLK', 'MASSACHUSETTS', 'DEMOCRATIC -
REPUBLICAN']
['7', 'ANDREW', 'JACKSON', 'WAXHAW', 'SOUTH CAROLINA', 'DEMOCRATIC']
['8', 'MARTIN', 'VAN BUREN', 'KINDERHOOK', 'NEW YORK', 'DEMOCRATIC']
['9', 'WILLIAM HENRY', 'HARRISON', 'BERKELEY', 'VIRGINIA', 'WHIG']
['10', 'JOHN', 'TYLER', 'CHARLES CITY COUNTY', 'VIRGINIA', 'WHIG']
['11', 'JAMES KNOX', 'POLK', 'MECKLENBURG COUNTY', 'NORTH CAROLINA', 'DEMOCRATIC']
['12', 'ZACHARY', 'TAYLOR', 'ORANGE COUNTY', 'VIRGINIA', 'WHIG']
['13', 'MILLARD', 'FILLMORE', 'CAYUGA COUNTY', 'NEW YORK', 'WHIG']
['14', 'FRANKLIN', 'PIERCE', 'HILLSBORO', 'NEW HAMPSHIRE', 'DEMOCRATIC']
['15', 'JAMES', 'BUCHANAN', 'COVE GAP', 'PENNSYLVANIA', 'DEMOCRATIC']
['16', 'ABRAHAM', 'LINCOLN', 'HODGENVILLE, HARDIN COUNTY', 'KENTUCKY', 'REPUBLICAN']
['17', 'ANDREW', 'JOHNSON', 'RALEIGH', 'NORTH CAROLINA', 'REPUBLICAN']
['18', 'ULYSSES SIMPSON', 'GRANT', 'POINT PLEASANT', 'OHIO', 'REPUBLICAN']
['19', 'RUTHERFORD BIRCHARD', 'HAYES', 'DELAWARE', 'OHIO', 'REPUBLICAN']
['20', 'JAMES ABRAM', 'GARFIELD', 'ORANGE, CUYAHOGA COUNTY', 'OHIO', 'REPUBLICAN']
```

Example

remote_cmd_facade.py

```
import paramiko
REMOTE_HOST = 'localhost'
REMOTE_USER = 'python'
REMOTE_PASSWORD = 'l0lz'

class RemoteConnection():
    def __init__(self, host, user, password):
        self._host = host
        self._user = user
        self._password = password
        self._connect()

    def _connect(self):
        self._ssh = paramiko.SSHClient()
        self._ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        self._ssh.connect(self._host, username=self._user, password=self._password)

    def run_command(self, command_line):
        stdin, stdout, stderr = self._ssh.exec_command(command_line)
        stdout_text = stdout.read()
        stderr_text = stderr.read()
        return stdout_text, stderr_text

    def __del__(self):
        self._ssh.close()

if __name__ == '__main__':
    rc = RemoteConnection(REMOTE_HOST, REMOTE_USER, REMOTE_PASSWORD)

    stdout, stderr = rc.run_command('whoami')
    print(stdout.decode(), '\n')

    stdout, stderr = rc.run_command('grep root /etc/passwd')
    print("STDOUT:", stdout.decode())
    print("STDERR:", stderr.decode())

    print('-' * 60)

    stdout, stderr = rc.run_command('grep root /etc/pizza')
    print("STDOUT:", stdout.decode())
    print("STDERR:", stderr.decode())
```

remote_cmd_facade.py

```
python


STDOUT: root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
_cvmsroot:*:212:212:CVMS Root:/var/empty:/usr/bin/false

STDERR:
------------------------------------------------------------
STDOUT:
STDERR: grep: /etc/pizza: No such file or directory
```

Flyweight

- Many small objects that share state
- E.g. characters in a word processor
- Use cases
    - Applications that need a large number of objects
    - Number of objects would drive up storage costs
    - Object state can be easily made external
    - Factor out duplicate state among many objects
    - Application doesn’t require unique objects

Participating objects: Flyweight Mixin, Flyweight, Unshared Flyweight, Flyweight Factory, Client

While it is useful to have objects at a fundamental level of granularity, it can be expensive in terms of memory usage, and possibly performance.

As an example, consider a word processor. It would be nice to represent individual characters as objects, but it would take large amounts of memory if there were separate objects for every character in a large document.

The Flyweight pattern allows you to create many instances of an objects, but instances can share state. In the word processor example, all instances of the letter "m" would really be the same object. Any context-dependent information is calculated by or passed in from the clients (i.e., objects using the FlyWeight objects.

There are several ways to implement a Flyweight in Python. In all cases, there is a repository of instances, and instances are only created via a factory. In the example, a decorator is used to turn a class into a Flyweight, and the class itself is the factory via the *call*method.

Each time an instance is needed, it is either drawn from the dictionary of instances, or created and added to the dictionary. The dictionary acts as a cache of the state of all instances.

NOTE For another example of using the FlyWeight decorator, see bigcat.py in the EXAMPLES folder

Command

- Encapsulate an operation
- Decoupled from the invoker
- Commands can be grouped
- Use cases – Parameter objects by task – Queue and execute tasks at different times – Support undo operations – Log changes to a system for replay – Transaction-based systems

Participating objects Command Mixin, Command, Client, Invoker, Receiver

The command pattern encapsulates an operation (or request). The code that executes the command is known as the invoker, and does not have to know the implementation details of the command object; it only needs to know the API, which can be inherited from an abstract base class to make sure the command object implements the necessary methods.

Individual command objects may be queued, and then executed in order at a specified time.

Example

command_pattern.py

```
#
from abc import ABCMeta, abstractmethod

class Command(metaclass=ABCMeta):
    '''Command "interface"'''

    @abstractmethod
    def execute(self): pass


class Fan:

    def start_rotate(self):
        print("Fan is rotating")

    def stop_rotate(self):
        print("Fan is not rotating")


class Light:

    def turn_on(self):
        print("Light is on")


    def turn_off(self):
        print("Light is off")


class Switch:

    def __init__(self, on, off):
        self.on_command = on
        self.off_command = off

    def on(self):
        self.on_command.execute()

    def off(self):
        self.off_command.execute()


class LightOnCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_on()


class LightOffCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_off()


class FanOnCommand(Command):
    def __init__(self, fan):
        self.fan = fan

    def execute(self):
        self.fan.start_rotate()

class FanOffCommand(Command):
    def __init__(self, fan):
        self.fan = fan

    def execute(self):
```

```python
        self.fan.stop_rotate()

if __name__ == '__main__':
    light = Light()
    light_on_command = LightOnCommand(light)
    light_off_command = LightOffCommand(light)

    light_switch = Switch(light_on_command, light_off_command)
    light_switch.on()
    light_switch.off()

    fan = Fan()
    fan_on_command = FanOnCommand(fan)
    fan_off_command = FanOffCommand(fan)

    fan_switch = Switch(fan_on_command, fan_off_command)
    fan_switch.on()
    fan_switch.off()
```

command_pattern.py

```
Light is on
Light is off
Fan is rotating
Fan is not rotating
```

Mediator

- Controls interaction among related objects
- Similar to facade, but two-way
- Objects don’t interact directly
- Use cases – Organize complex interdependence of objects – Reuse an object that interacts with other objects – Configure multi-object behavior

Participants

```
Mediator Mixin, Mediator, Colleague Classes
```

A mediator encapsulates interaction among related classes. Mediators are similar to facades, but are two-way, where a facade is generally one-way. A mediator knows the behavior of multiple objects, so that every object doesn’t have to know about every other object. It reduces the coupling within a system that has many objects.

Chapter 8 Exercises

Exercise 8-1 (deco_timer.py)

Create a decorator named functiontimer that can be applied to a function and reports the number of seconds (fractional) it takes for the function to execute. You can get the time from the time.time() function. (i.e., import time, and call the time() function from it.)

Just subtract the start time from the end time and report it. You don’t need any further calculations.

Write some functions and test your decorator on them.

Chapter 9: Packaging

Objectives

- Learn about packages vs apps
- Write setup files
- Understand the types of wheels
- Know when to create a non-wheel distribution
- Create a reusable package
- Configure dependencies
- Configure executable scripts
- Distribute and deploy packages

NOTE This chapter is a work in progress. There may be missing parts, or errors in code or links.

How to package (the old way)

- Start with standard layout
- Add configuration files
- Use setuptools or build tools to build package

To create a Python package, start with a good standard project layout. Add configuration and other files to the top level of the project.

Create the `setup.py` configuration module (which calls the **setup** function.

Use **setuptools** or **build** to build the package as an installable source or binary package.

Package files with `setup.py`

- README.rst reStructuredText setup doc
- MANIFEST.in list of all files in package
- LICENSE s/w license
- setup.py controls packaging and installation

There are four files to create in the package.

README.rst is a typical README file, that describes what the package does in brief. It is not the documentation for the package. It should tell how to install the package. Is is usual to use reStructuredText for this file, hence the .rst extension.

MANIFEST.in is a list of all the non-Python files in the package, such as templates, CSS files, and images. It should also include LICENSE and README.rst.

LICENSE is a file describing the license under which you're releasing the software. This is less important for in-house apps, but essential for apps that are distributed to the public.

The most import file is **setup.py**. This is a Python script that uses setuptools to control how your application is packaged for distribution, and how it is installed by users.

setup.py contains a call to the setup() function; named options configure the details of your package.

TIP The following link has some great suggestions for laying out the files in a package: https://blog.ionelmc.ro/2014/05/25/python-packaging/

Overview of setuptools

- Creates distributable files
- Can be used for modules or packages
- Many distribution formats
- Configuration in setup.py

The key to using setuptools is setup.py, the configuration file. This tells setuptools what files should go in the module, the version of the application or package, and any other configuration data.

The steps for using setuptools are:

- write a setup script (setup.py by convention)
- (optional) write a setup configuration file
- create a source, wheel, or specialized built distributions

The entire process is described at https://packaging.python.org/distributing/.

See https://packaging.python.org/glossary/#term-built-distribution for a glossary of packaging and distribution terms.

Preparing for distribution

- Organize files and folders
- Create setup.py

The first step is to create setup.py in the package folder.

setup.py is a python script that calls the setup() function from setuptools with keyword (named) arguments that describe your module.

For modules, use the py_modules keyword; for packages, use the packages.

There are many other options for fine-tuning the distribution.

Your distribution should also have a file named README or README.txt which can be a brief description of the distribution and how to install its module(s).

You can include any other files desired. Many developers include a LICENSE.txt which stipulates how the distribution is licensed.

Table 13. Keyword arguments for setup() function

Keyword	Description
`author`	Package author's name
`author_email`	Email address of the package author
`classifiers`	A list of classifiers to describe level (alpha, beta, release) or supported versions of Python
`data_files`	Non-Python files needed by distribution from outside the package
`description`	Short, summary description of the package
`download_url`	Location where the package may be downloaded
`entry_points`	Plugins or scripts provided in the package. Use key console_scripts to provide standalone scripts.
`ext_modules`	Extension modules needing special handling
`ext_package`	Package containing extension modules
`install_requires`	Dependencies. Specify modules your package depends on.
`keywords`	Keywords that describe the project
`license`	license for the package
`long_description`	Longer description of the package
`maintainer`	Package maintainer's name
`maintainer_email`	Email address of the package maintainer
`name`	Name of the package
`package_data`	Additional non-Python files needed from within the package
`package_dir`	Dictionary mapping packages to folders
`packages`	List of packages in distribution
`platforms`	A list of platforms
`py_modules`	List of individual modules in distribution
`scripts`	Configuration for standalone scripts provided in the package (but entry_points is preferred)
`url`	Home page for the package
`version`	Version of this release

Creating a source distribution

- Use setup.py with -sdist option
- Creates a platform-neutral distribution
- Distribution has its own setup.py

Run setup.py with your version of python, specifying the -sdist option. This will create a platform-independent source distribution. python setup.py sdist

```
ls -l dist
total 4
-rw-rw-r-- 1 jstrick jstrick 633 2012-01-11 07:49 temperature-1.2.tar.gz

tar tzvf dist/temperature-1.2.tar.gz
drwxrwxr-x jstrick/jstrick   0 2012-01-11 00:26 temperature-1.2/
-rw-rw-r-- jstrick/jstrick 256 2012-01-11 00:26 temperature-1.2/PKG-INFO
-rw-rw-r-- jstrick/jstrick 285 2012-01-10 13:36 temperature-1.2/setup.py
-rw-r--r-- jstrick/jstrick 342 2012-01-10 07:52 temperature-1.2/temperature.py
```

To install a source distribution, extract it into any directory and cd into the root (top) of the extracted file structure. Execute the following command:

```
python setup.py install
```

Example

temperature/setup.py

```
from setuptools import setup

# note -- this file is not used when building with pyproject.toml

if __name__ == "__main__":
    setup()
```

```
cd temperature
python setup.py sdist
running sdist
running egg_info
writing temperature.egg-info/PKG-INFO
writing top-level names to temperature.egg-info/top_level.txt
writing dependency_links to temperature.egg-info/dependency_links.txt
reading manifest file 'temperature.egg-info/SOURCES.txt'
writing manifest file 'temperature.egg-info/SOURCES.txt'
warning: sdist: standard file not found: should have one of README, README.rst,
README.txt

running check
creating temperature-1.0.0
creating temperature-1.0.0/temperature.egg-info
making hard links in temperature-1.0.0...
hard linking setup.py -> temperature-1.0.0
hard linking temperature.py -> temperature-1.0.0
hard linking temperature.egg-info/PKG-INFO -> temperature-1.0.0/temperature.egg-info
hard linking temperature.egg-info/SOURCES.txt -> temperature-1.0.0/temperature.egg-info
hard linking temperature.egg-info/dependency_links.txt -> temperature-
1.0.0/temperature.egg-info
hard linking temperature.egg-info/top_level.txt -> temperature-1.0.0/temperature.egg-info
Writing temperature-1.0.0/setup.cfg
Creating tar archive
removing 'temperature-1.0.0' (and everything under it)
```

```
tree dist
dist
└── temperature-1.2.tar.gz
```

Creating wheels

- 3 kinds of wheels
    - Universal wheels (pure Python; python 2 *and* 3 compatible
    - Pure Python wheels (pure Python; Python 2 *or* 3 compatible
    - Platform wheels (Platform-specific; binary)

A wheel is prebuilt distribution. Wheels can be installed with pip.

A Universal wheel is a pure Python package (no extensions) that can be installed on either Python 2 or Python 3. It has to have been carefully written that way.

A Pure Python wheel is a pure Python package that is specific to one version of Python (either 2 or 3). It can only be installed by a matching version of pip.

A Platform wheel is a package that has extensions, and thus is platform-specific.

Example

```
python setup.py bdist_wheel
running bdist_wheel
running build
running build_py
creating build
creating build/lib
copying temperature.py -> build/lib
installing to build/bdist.macosx-10.6-x86_64/wheel
running install
running install_lib
creating build/bdist.macosx-10.6-x86_64
creating build/bdist.macosx-10.6-x86_64/wheel
copying build/lib/temperature.py -> build/bdist.macosx-10.6-x86_64/wheel
running install_egg_info
running egg_info
writing temperature.egg-info/PKG-INFO
writing top-level names to temperature.egg-info/top_level.txt
writing dependency_links to temperature.egg-info/dependency_links.txt
reading manifest file 'temperature.egg-info/SOURCES.txt'
writing manifest file 'temperature.egg-info/SOURCES.txt'
Copying temperature.egg-info to build/bdist.macosx-10.6-x86_64/wheel/temperature-1.0.0
-py2.7.egg-info
running install_scripts
creating build/bdist.macosx-10.6-x86_64/wheel/temperature-1.0.0.dist-info/WHEEL
```

```
tree dist
dist
├── temperature-1.2-py3-none-any.whl
└── temperature-1.2.tar.gz
```

Creating other built distributions

- Use setup.py with -bdist –format=format
- Creates platform-specific distributions
- Common Unix formats: rpm, deb, tgz
- Common Windows formats: msi, exe

For the convenience of the end-user, you can create "built" distributions, which are ready to install on specific platforms. These are built with the bdist argument to setup.py, plus a --format=format option to indicate the target platform.

```
python setup.py bdist --format=rpm
```

```
tree dist
dist
├── temperature-1.2-py3-none-any.whl
├── temperature-1.2.macosx-10.9-x86_64.tar
├── temperature-1.2.macosx-10.9-x86_64.zip
└── temperature-1.2.tar.gz
```

Installing a package

- Use pip for wheels
- Use setup.py for source

One of the advantages of wheels is that they make installing packages easier. You can just use

pip install *package*.whl

If you have a source distribution, extract the source in any convenient location (this is not permanent) and cd to the top-level folder. Use the following command:

```
python setup.py install
```

To install in the default location.

Use

```
python setup.py install –prefix=ALTERNATE-DIR
```

to install under an alternate prefix.

Using Cookiecutter

- Create standard layout
- Developed for Django
- Very flexible

cookiecutter is a utility written by Audrey and Roy Greenfeld to make it easy to replicate a standard setup for Django. The cookiecutter command prompts you for information, then creates the project folder.

It uses a cookiecutter *template*, which is a folder, to create the new project. There are many templates on **github** to choose from, and you can easily create your own.

The script copies the template layout (all folders and files) to a new folder which is the "slug" (short name) of your project. It inserts your project name in the appropriate places.

cookiecutter home page: https://github.com/audreyr/cookiecutter
cookiecutter docs: https://cookiecutter.readthedocs.io

Package files the new way

- Create `pyproject.toml`
- `setup.py` and `setup.cfg` not needed
  - (probably)
- Use `build` to build the package

The **modern** way to build a Python package is using the `pyproject.toml` config file. The specifications that support this are specified in **PEP 518** and **PEP 621**.

The **TOML** format is similar to `.ini` files, but adds some features.

The first part of the file is required. It tells the `build` program what tools to use.

```
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"
```

the rest of the file is not needed if you are also using `setup.cfg` and `setup.py`. However, there is a trend away from using those legacy files, as all the data needed for building the package, installing it, and uploading it to **PyPI** can be contained in `pyproject.toml`, and can be used by any build *backend*.

```
[project]
name = "wombatfun"
version = "1.0.0"
authors = [
    { name="Author Name", email="jstrickler@gmail.com" },
]
description = "Short Description of the Package"
readme = "README.rst"
requires-python = ">=3.0"
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]

dependencies = [
    'requests[security] < 3',
]
```

Building the project

- `python -m build`
- Creates `dist` folder
- Binary distribution
    - `package-version.whl`
- Source distribution
    - `package-version.tar.gz`

To build the project, use

```
python -m build
```

This will create the wheel file (binary distribution) and a gzipped tar file (source distribution) in a folder named `dist`.

Editable installs

- Use `pip install -e package`
- Puts a link in library folder
- Allows testing as though module is installed

When using a `src` (or other name) folder for your codebase and `tests` for your test scripts, the tests need to find your package. While you could put the path to the `src` folder in PYTHONPATH, the best practice is to do an *editable install*.

This is an install that uses the path to your development folder. It achieves this by using a virtual environment. Then you can run your tests after making changes to your code, without having to reinstall the package.

From the top level folder of the project, type the following (you do not have to build the distribution for this step).

```
pip install -e .
```

Then you can just say

```
pytest -v
```

to run the tests in verbose mode, or any other variation of `pytest`.

Combining setup and build

- Provide backward compatibility
- TOML approach can have glitches

to be completed

For more information

Python Packaging User Guide

https://packaging.python.org/en/latest/

Distributing Python Modules

https://docs.python.org/3/distributing/index.html

setuptools Quickstart

https://setuptools.pypa.io/en/latest/userguide/quickstart.html

Thoughts on the Python packaging ecosystem

https://pradyunsg.me/blog/2023/01/21/thoughts-on-python-packaging/

THE BASICS OF PYTHON PACKAGING IN EARLY 2023

https://drivendata.co/blog/python-packaging-2023

Structuring Your Project (from The Hitchhiker's Guide to Python)

https://docs.python-guide.org/writing/structure/

Chapter 9 Exercises

Exercise 9-1 (president/*)

Implement a distributable package from the **President** class created in the chapter on object-oriented programming.

Create a wheel file, then try to install it with **pip**.

Chapter 10: Functional Tools

Objectives

- Conceptualize higher-order functions
- Learn what's in functools and itertools
- Transform data with map/reduce
- Chain multiple iterables together
- Implement function overloading with single dispatch
- Use iterative tools to work with iterators
- Chain multiple iterators together
- Compute combinations and permutations
- Zip multiple iterables with default values

Higher-order functions

- Functions that operate on (or return) functions

Higher-order functions operate on or return functions. The most traditional higher-order functions in other languages are `map()` and `reduce()`. These are used for functional programming, which prevents side effects (modifying data outside the function).

Some languages make it difficult or impossible to pass functions into other functions, but in Python, since functions are first-class objects, it is easy.

Higher-order functions may be nested, to have multiple transformations on data.

Lambda functions

- Inline anonymous functions
- Contain only parameters and return value
- Useful as predicates in higher-order functions

A **lambda** function is an inline anonymous function declaration. It evaluates as a function object, in the same way that `def function(…):` … does.

A lambda declaration has only parameters and the return value. Blocks are not allowed, nor is the **return** keyword.

Lambdas are useful as predicates (callbacks) in higher-order functions.

Example

higher_order_functions.py

```
#

fruits = ["pomegranate", "cherry", "apricot", "date", "apple",
          "lemon", "kiwi", "orange", "lime", "watermelon", "guava",
          "papaya", "fig", "pear", "banana", "tamarind", "persimmon",
          "elderberry", "peach", "blueberry", "lychee", "grape"]


def process_list(alist, func):  # Define a function that accepts a list and a passed-in
function (AKA callback)
    new_list = []
    for item in alist:
        new_list.append(func(item))  # Call the callback function on one item of the
passed-in list
    return new_list


f1 = process_list(fruits, str.upper)  # Call process_list() with str.upper as the
callback
print(f1, "\n")

f2 = process_list(fruits, lambda s: s[0].upper())  # Call process_list() with a lambda
function as the callback
print(f2, "\n")

f3 = process_list(fruits, len)  # Call process_list() with the builtin function len() as
the callback()
print(f3, "\n")

total_length = sum(process_list(fruits, len))  # Pass the result of process_list() to the
builtin function sum() to sum all the values in the returned list

print(total_length, "\n")
```

higher_order_functions.py

```
['POMEGRANATE', 'CHERRY', 'APRICOT', 'DATE', 'APPLE', 'LEMON', 'KIWI', 'ORANGE', 'LIME',
'WATERMELON', 'GUAVA', 'PAPAYA', 'FIG', 'PEAR', 'BANANA', 'TAMARIND', 'PERSIMMON',
'ELDERBERRY', 'PEACH', 'BLUEBERRY', 'LYCHEE', 'GRAPE']

['P', 'C', 'A', 'D', 'A', 'L', 'K', 'O', 'L', 'W', 'G', 'P', 'F', 'P', 'B', 'T', 'P',
'E', 'P', 'B', 'L', 'G']

[11, 6, 7, 4, 5, 5, 4, 6, 4, 10, 5, 6, 3, 4, 6, 8, 9, 10, 5, 9, 6, 5]

138
```

The operator module

- Provides operators as functions
- Use rather than simple lambdas

The `operator` module provides functional versions of Python’s standard operators. This saves the trouble of creating trivial lambda functions.

Instead of

```
lambda x, y: x + y
```

You can just use

```
operator.add
```

Both of these functions take two operators and add them, return the result.

Example

operator_module.py

```python
from operator import add


a = 10
b = 15

print("a + b: {}".format(a + b))  # Add with add operator
print("add(a, b): {}".format(add(a, b)))  # Add with add function
```

operator_module.py

```
a + b: 25
add(a, b): 25
```

The functools module

- Included in standard library
- Supports higher-order programming
- Many standard higher-order functions

The `functools` module provides tools for higher-ordering programming, which means functions that operate on, or return functions (some do both). Functional programming avoids explicit loops.

`map()` and `reduce()` are two functions that are the basis of many functional algorithms. `map()` is a builtin function. `reduce()` *was* builtin in Python 2, but in Python 3 must be imported from `functools`.

NOTE Most of these functional algorithms can also be implemented with list comprehensions or generator expressions, which many people find easier to both read and write.

map()

- Applies function to every element of iterable
- Syntax
  - `map(function, iterable)`

map() returns a virtual list (i.e., a generator) created by applying a function to every element of an iterable.

`map(func, list)` returns an iterator like `[func(list[0]), func(list[1]), func(list[2], …)]`

The first argument to `map()` is a function that takes one argument. Each element of the iterable is passed to the function, and the return value is added to the result generator.

Example

using_map.py

```
#

strings = ['wombat', 'koala', 'kookaburra', 'blue-ringed octopus']

result = [s.upper() for s in strings]  # Using a list comprehension, which is usually
simpler than map()
print(result)

result = list(map(str.upper, strings))  # Using map to copy list to upper case
print(result)

result = list(map(len, strings))  # Using map to get list of string lengths
print(result)
```

using_map.py

```
['WOMBAT', 'KOALA', 'KOOKABURRA', 'BLUE-RINGED OCTOPUS']
['WOMBAT', 'KOALA', 'KOOKABURRA', 'BLUE-RINGED OCTOPUS']
[6, 5, 10, 19]
```

reduce()

- Imported from `functools`
- Applies function to every element plus previous result

`reduce()` returns the value created by applying a function to every element of a list *and* the previous function result. `reduce()` must be imported from the `functools` module.

`reduce(func, list)` returns `func(list[n], func(list[n-1], func(list[2]…func(list[0]))))`

A third argument to `reduce()` provides the initial value. By default, the initial value is the first element of the list.

Other functions such as `sum()` or `str.join()` can be defined in terms of `map()` or `reduce()`.

The **mapreduce** approach to massively parallel processing, such as Hadoop, was inspired by map() and reduce().

Example

using_reduce.py

```
#
from operator import add, mul
from functools import reduce

values = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# sum()
result = reduce(add, values) # add values in list (initial value defaults to 0)
print("result is", result)

# sum() + 1000
result = reduce(add, values, 1000)  # add values in list (initial value is 1000)
print("result is", result)

# product
result = reduce(mul, values)  # multiply all values together (initial value is 1,
otherwise product would be 0)
print("result is", result)

strings = ['fi', 'fi', 'fo', 'fum']

# join
result = reduce(add, strings, "") # concatenate strings (initial value is empty string;
each string in iterable added to it)
print("result is", result)

# join + upper case
result = reduce(add, map(str.upper, strings), "")  # same, but make strings upper case
print("result is", result)
```

using_reduce.py

```
result is 550
result is 1550
result is 3628800000000000
result is fififofum
result is FIFIFOFUM
```

Partial functions

- Some arguments filled in
- Create desired signature

Partial functions are wrappers that have some arguments already filled in for the "real" function. This is especially useful when creating callback functions. It is also nice for creating customized functions that rely on functions from the standard library.

While you can create partial functions by hand, using closures, the `partial()` function (from the `functools` module) simplifies creating such a function.

The arguments to `partial()` are the function and one or more arguments. It returns a new function object, which will call the specified function and pass in the provided argument(s).

Example

partial_functions.py

```python
#
import re

from functools import partial

count_by = partial(range, 0, 25)  # create partial function that "preloads" range() with
arguments 0 and 25

print((list(count_by(1))))  # call partial function with parameter, 0 and 25
automatically passed in
print((list(count_by(3))))  # call partial function with parameter, 0 and 25
automatically passed in
print((list(count_by(5))))  # call partial function with parameter, 0 and 25
automatically passed in
print()

has_a_number = partial(re.search, r'\d+')  # create partial function that embeds pattern
in re.search()

strings = [
    'abc', '123', 'abc123', 'turn it up to 11', 'blah blah'
]

for s in strings:
    print("{}:".format(s), end=' ')
    if has_a_number(s): # call re.search() with specified pattern
        print("YES")
    else:
        print("NO")
```

partial_functions.py

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24]
[0, 3, 6, 9, 12, 15, 18, 21, 24]
[0, 5, 10, 15, 20]

abc: NO
123: YES
abc123: YES
turn it up to 11: YES
blah blah: NO
```

Single dispatch

- Provides generic functions
- Emulates function/method overloading
- Functions registered by signature

The `singledispatch` class provides *generic functions*. A generic function is defined once with the `singledispatch()` decorator, then other functions may be registered to it.

To register a function, decorate it with `@original_function.register(type)`. Then, when the original function is passed an argument of type type as its first parameter, it will call the registered function instead. If no registered functions are found for the type, it will call the original function.

This can of course, be done manually by checking argument types, then calling other methods, but using `singledispatch` is much less cumbersome.

To check which function would be chosen for a given type, use the `dispatch()` method of the original function. To get a list of all registered functions, use the registry attribute.

The `singledispatch` module was added to the standard library beginning with Python 3.4.

Because it only works for the first parameter, `singledispatch` was not in the past useful for class methods, but starting with Python 3.8, the `functools.singledispatchmethod` class will provide single dispatch for methods.

Example

single_dispatch.py

```
from singledispatch import singledispatch
from io import TextIOWrapper

@singledispatch
def xopen(source, mode="r"):  # define generic function that is actually called
    source_type = type(source).__name__
    raise TypeError("Invalid arg: must be file, str, or bytes, not {}
".format(source_type))

@xopen.register(TextIOWrapper)  # define handler for TextIOWrapper type (normal text
files)
def _(fileobj):
    return fileobj

@xopen.register(str)  # define handler for string type
def _(str, mode="r"):
    return open(str, mode)


@xopen.register(bytes)  # define handler for bytes type
def _(bytes, mode="r"):
    return open(bytes.decode(), mode)


mary_in = open('../DATA/mary.txt')  # open a file and get a file object (type
TextIOWrapper)

for x in mary_in, '../DATA/mary.txt', b'../DATA/mary.txt', 52, ['a', 5]:
    try:
        file_in = xopen(x)  # call single dispatch function -- correct handler will
automatically be called
        result = file_in.read()
        print("Length: {}".format(len(result)))
    except TypeError as err:
        print('ERROR:', err)

print('-' * 60)
print(xopen.dispatch(str), "\n")  # show handler function for str

for arg_type, func in xopen.registry.items():
    print("{:30s} {}".format(str(arg_type), func))  # show functions for each registered
type
```

single_dispatch.py

```
Length: 108
Length: 108
Length: 108
ERROR: Invalid arg: must be file, str, or bytes, not int
ERROR: Invalid arg: must be file, str, or bytes, not list
------------------------------------------------------------
<function _ at 0x7faac00a9ca0>

<class 'object'>               <function xopen at 0x7faad00e8040>
<class '_io.TextIOWrapper'>    <function _ at 0x7faac00a9c10>
<class 'str'>                  <function _ at 0x7faac00a9ca0>
<class 'bytes'>                <function _ at 0x7faac00a9d30>
```

The itertools module

- Tools to help with iteration
- Provide many types of iterators

The `itertools` module provides many different iterators. Some of the tools work on existing iterators, while others create them from scratch.

These tools work well with functions from the operator module, as well as interacting with the functional tools described earlier.

Many of the constructors in `itertools` were inspired by Haskell, SML, and APL.

See **iterable_recipes.py** in the EXAMPLES folder for a group of utility functions that use many of the routines in this chapter.

Infinite iterators

- Iterate infinitely, or specified number of times
- Cycle over or repeat values

Infinite iterators return iterators that will iterate a specified number of times, or infinitely. These iterators are similar to generators; they do not keep all the values in memory.

`islice()` selects a slice of an iterator. You can specify an iterable and up to 3 slice arguments – start, stop, and increment, similar to the the slice arguments of a list. If just stop is provided, it stops the iterator after than many values.

`count()` is similar to `range()`. It provides a sequence of numbers with a specified increment. The big difference is that there is no end condition, so it will increment forever. You will need to to test the values and stop, or use islice().

`cycle()` loops over an iterable repeatedly, going back to the beginning each time the end is reached.

`repeat()` repeats a value infinitely, or a specified number of times.

Example

infinite_iterators.py

```
#
from itertools import islice, count, cycle, repeat

for i in count(0, 10):  # count by tens starting at 0 forever
    if i > 50:
        break  # without a check, will never stop
    print(i, end=' ')
print("\n")

for i in islice(count(0, 10), 6):  # saner, using islice to get just the first 6 results
    print(i, end=' ')
print("\n")

giant = ['fee', 'fi', 'fo', 'fum']

for i in islice(cycle(giant), 10):  # cycle over values in list forever (use islice to
stop)
    print(i, end=' ')
print("\n")

for i in repeat('tick', 10):  # repeat value 10 times (default is repeat forever)
    print(i, end=' ')
print("\n")
```

infinite_iterators.py

```
0 10 20 30 40 50

0 10 20 30 40 50

fee fi fo fum fee fi fo fum fee fi

tick tick tick tick tick tick tick tick tick tick
```

Extended iteration

- Extended iteration over multiple objects
- Truncated iteration over a list

Another group of iterator functions provides extended iteration.

`chain()` takes two or more iterables, and treats them as a single iterable.

To chain the elements of a single iterable together, use `chain.from_iterable()`.

`dropwhile()` skips leading elements of an iterable until some condition is reached. `takewhile()` stops iterating when some condition is reached.

Example

extended_iteration.py

```
#
from itertools import chain, takewhile, dropwhile

spam = ['alpha', 'beta', 'gamma']
ham = ['delta', 'epsilon', 'zeta']

for letter in chain(spam, ham):  # treat spam and ham as a single iterable
    print(letter, end=' ')
print("\n")

eggs = [spam, ham]

for letter in chain.from_iterable(eggs):  # treat all elements of eggs as a single
iterable
    print(letter, end=' ')
print("\n")

fruits = ["pomegranate", "cherry", "apricot", "date", "apple",
          "lemon", "kiwi", "orange", "lime", "watermelon", "guava",
          "papaya", "fig", "pear", "banana", "tamarind", "persimmon",
          "elderberry", "peach", "blueberry", "lychee", "grape"]

for fruit in takewhile(lambda f: len(f) > 4, fruits):  # iterate over elements of fruits
as long as length of current item > 4
    print(fruit, end=' ')
print("\n")

for fruit in takewhile(lambda f: f[0] != 'k', fruits):  # iterate over elements of
fruits as long as fruit does not start with 'k'
    print(fruit, end=' ')
print("\n")

values = [5, 18, 22, 31, 44, 57, 59, 61, 66, 70, 72, 78, 90, 99]

for value in dropwhile(lambda f: f < 50, values):  # skip over elements of values as long
as value is < 50, then iterate over all remaining elements
    print(value, end=' ')
print("\n")
```

extended_iteration.py

```
alpha beta gamma delta epsilon zeta

alpha beta gamma delta epsilon zeta

pomegranate cherry apricot

pomegranate cherry apricot date apple lemon

57 59 61 66 70 72 78 90 99
```

Grouping

- Groups consecutive elements by value
- Input must be sorted

The `groupby()` function groups consecutive elements of an iterable by value. As with sorted(), the value my be determined by a key function. This is similar to sort -u or the uniq commands in Linux.

groupby() returns an iterable of subgroups, which can then be converted to a list or iterated over. Each subgroup has a key, which is the common value, and an iterable of the values for that key.

For a more real-life example of grouping, see `group_dates_by_week.py` in the EXAMPLES folder.

Example

groupby_examples.py

```python
from itertools import groupby

with open('../DATA/words.txt') as words_in:  # open file for reading
    all_words = (w.rstrip() for w in words_in)  # create generator of all words, stripped of the trailing '

    g = groupby(all_words, key=lambda e: e[0])  # create a groupby() object where the key is the first character in the word

    counts = {letter: len(list(wlist)) for letter, wlist in g}  # make a dictionary where the key is the first character, and the value is the number of words that start with that character; groupby groups all the words, then len() counts the number of words for that character

sorted_letters = sorted(counts.items(), key=lambda e: e[1], reverse=True)  # sort the counts dictionary by value (i.e., number of words, not the letter itself) into a list of tuples
for letter, count in sorted_letters:  # loop over the list of tuples and print the letter and its count
    print(letter, count)

print()
print("Total words counted:", sum(counts.values()))  # sum all the individual counts and print the result
```

groupby_examples.py

```
s 19030
c 16277
p 14600
a 10485
r 10199
d 10189
m 9689
b 9325
t 8782
e 7086
i 6994
f 6798
h 6333
o 5858
g 5599
u 5095
l 5073
n 4411
w 3727
v 2741
k 1734
j 1378
q 827
y 552
z 543
x 137

Total words counted: 173462
```

Combinatoric generators

- Provides products, combinations and permutation
- Can specify max length of sub-sequences

Several functions provide products, combinations, and permutations.

product() return an iterator with the Cartesian product of two iterables.

combinations() returns the unique n-length combinations of an iterator.

permutations() returns all n-length sub-sequences of an iterator.

If the length is not specified, all sub-sequences are returned.

Example

combinations_permutations.py

```
#
from itertools import product, permutations, combinations

SUITS = 'CDHS'
RANKS = '2 3 4 5 6 7 8 9 10 J Q K A'.split()

cards = product(SUITS, RANKS)  # Cartesian product (match every item in one list to every
item in the other list)
print(list(cards), '\n')

cards = [r + s for r, s in product(SUITS, RANKS)]  # reverse order and concatenate
elements using list comprehension
print(cards, '\n')

giant = ['fee', 'fi', 'fo', 'fum']

result = combinations(giant, 2)  # all distinct combinations of 4 items taken 2 at a time
print(list(result), "\n")

result = permutations(giant, 2)  # all distinct permutations of 4 items taken 2 at a time
print(list(result), "\n")
```

combinations_permutations.py

```
[('C', '2'), ('C', '3'), ('C', '4'), ('C', '5'), ('C', '6'), ('C', '7'), ('C', '8'),
('C', '9'), ('C', '10'), ('C', 'J'), ('C', 'Q'), ('C', 'K'), ('C', 'A'), ('D', '2'),
('D', '3'), ('D', '4'), ('D', '5'), ('D', '6'), ('D', '7'), ('D', '8'), ('D', '9'), ('D',
'10'), ('D', 'J'), ('D', 'Q'), ('D', 'K'), ('D', 'A'), ('H', '2'), ('H', '3'), ('H',
'4'), ('H', '5'), ('H', '6'), ('H', '7'), ('H', '8'), ('H', '9'), ('H', '10'), ('H',
'J'), ('H', 'Q'), ('H', 'K'), ('H', 'A'), ('S', '2'), ('S', '3'), ('S', '4'), ('S', '5'),
('S', '6'), ('S', '7'), ('S', '8'), ('S', '9'), ('S', '10'), ('S', 'J'), ('S', 'Q'),
('S', 'K'), ('S', 'A')]

['C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9', 'C10', 'CJ', 'CQ', 'CK', 'CA', 'D2',
'D3', 'D4', 'D5', 'D6', 'D7', 'D8', 'D9', 'D10', 'DJ', 'DQ', 'DK', 'DA', 'H2', 'H3',
'H4', 'H5', 'H6', 'H7', 'H8', 'H9', 'H10', 'HJ', 'HQ', 'HK', 'HA', 'S2', 'S3', 'S4',
'S5', 'S6', 'S7', 'S8', 'S9', 'S10', 'SJ', 'SQ', 'SK', 'SA']

[('fee', 'fi'), ('fee', 'fo'), ('fee', 'fum'), ('fi', 'fo'), ('fi', 'fum'), ('fo',
'fum')]

[('fee', 'fi'), ('fee', 'fo'), ('fee', 'fum'), ('fi', 'fee'), ('fi', 'fo'), ('fi',
'fum'), ('fo', 'fee'), ('fo', 'fi'), ('fo', 'fum'), ('fum', 'fee'), ('fum', 'fi'),
('fum', 'fo')]
```

Chapter 10 Exercises

Exercise 10-1 (sum_of_values.py)

Read in the data from float_values.txt and print out the sum of all values. Do this with functional tools – there should be no explicit loops in your code.

TIP use reduce() + operator.add on the file object.

Exercise 10-2 (pres_by_state_func.py)

Using presidents.txt, print out a list of the number of presidents from each state.

TIP Use map() + lambda to split lines from presidents.txt on the 7th field, then use groupby() on that.

Exercise 10-3 (count_all_lines.py)

Count all of the lines in all the files specified on the command line without using any loops.

TIP use map() + chain.from_iterable() to create an iterable of all the lines, then use reduce to count them.

Chapter 11: Container Classes

Objectives

- Recap builtin container types
- Discover extra container types in standard library
- Save space with array objects
- Create custom variations of container types

Container classes

- Contain all elements in memory objects
- Elements are objects or key/object pairs
- Have a length
- Are indexable and iterable

Container classes are objects that can hold multiple objects. All of the objects contained are kept in memory. Containers can be indexed, and can be iterated over. All containers have a length, which is the number of elements.

Builtin containers

- list, tuple array-like
- dict dictionary
- set, frozenset set

Several container types are builtin.

There are two array-like containers – list and tuple. A list is a dynamic array, while a tuple is more like a struct or a record types. Both are array-like, meaning they have a length, can be indexed, and can be sliced.

The dict type is a dictionary of key/value pairs. In some languages, dictionaries are called hashes, or even more exotic names. Dictionaries are dynamic. Keys must be unique. The set type contains unique values. Elements may be added to and deleted from a set. A frozenset is a readonly set.

Example

builtin_containers.py

```
alist = ['alpha', 'beta', 'gamma', 'eta', 'zeta']
atuple = ('123 Elm Street', 'Toledo', 'Ohio')
adict = {'alpha': 5, 'beta': 10, 'gamma': 15}
aset = {'alpha', 'beta', 'gamma', 'eta', 'zeta'}

print(alist[0], atuple[0], adict['alpha'])
print(alist[-1], atuple[-1])
print(alist[:3], alist[3:], alist[2:5], alist[-2:])
print('alpha' in alist, 'Ohio' in atuple, 'gamma' in adict, 'zeta' in aset)
print(len(alist), len(atuple), len(adict), len(aset))
print(alist.count('alpha'), atuple.count('Ohio'))
```

builtin_containers.py

```
alpha 123 Elm Street 5
zeta Ohio
['alpha', 'beta', 'gamma'] ['eta', 'zeta'] ['gamma', 'eta', 'zeta'] ['eta', 'zeta']
True True True True
5 3 3 5
1 1
```

Containers in the standard library

- Many containers in the collections module
- Variations of lists, tuples, and dicts

The `collections` module provides several extra container types. In general, these are variations on lists, tuples, and dicts.

These types are:

- Counter – dictionary designed for counting
- defaultdict – dictionary with default value for missing keys
- deque – array optimized for access at ends only
- namedtuple – tuple with named elements
- OrderedDict – dictionary that remembers order elements were inserted

A **Counter** is a dict with a default value of 0, so values may be incremented without check to see whether the key exists. In addition, counter provides the *n* most common elements counted.

A **defaultdict** is a dict that provides a default value for missing keys. When the defaultdict is created, you pass in a function that provides that default value. If you need a constant value, such as 0 or "", you can use a lambda expression for the function.

A **deque** (pronounced "deck") is a double-ended queue. It is optimized for inserting and removing from the ends, and is much more efficient than a **list** for this purpose. The deque can be initialized with any iterable.

A **namedtuple** is a tuple with specified field names. In addition to accessing fields as *tuple*[0], you can access them as *tuple*.*field_name*. Named tuples map very well to C **structs**.

An **OrderedDict** is a dictionary which preserves order. Any iteration over the dictionary's keys or values is guaranteed to be in the same order that the elements were added.

NOTE Starting with Python 3.6, all dictionaries preserve order, making OrderedDicts obsolete. (Of course legacy code may still use them, and some existing modules in the stdlib use or return them).

Example

stdlib_containers.py

```
from collections import Counter, defaultdict, deque, namedtuple, OrderedDict

# Counter
with open('../DATA/words.txt') as words_in:
    all_words = [line[0] for line in words_in]
    word_counter = Counter(all_words)  # Count list of words by passing iterable of words
to Counter instance


print(word_counter.most_common(10))  # Counter.most_common() return n most common
occurrences
print('-' * 60)

# defaultdict
fruits = ["pomegranate", "cherry", "apricot", "date", "apple",
          "lemon", "kiwi", "orange", "lime", "watermelon", "guava",
          "papaya", "fig", "pear", "banana", "tamarind", "persimmon",
          "elderberry", "peach", "blueberry", "lychee", "grape"]

fruit_by_first = defaultdict(list)  # Create default dict whose default value is a new
list object

for fruit in fruits:
    fruit_by_first[fruit[0]].append(fruit)  # Append fruit to dictionary element whose
key is the first letter and whose value is a list

for letter, fruits in sorted(fruit_by_first.items()):
    print(letter, fruits)
print('-' * 60)

# deque
d = deque()  # Create an empty deque
for c in 'abcdef':
    d.append(c)   # Append to the deque
print(d)
for c in 'ghijkl':
    d.appendleft(c)  # Prepend to the deque
print(d)
d.extend('mno')  # Extend the deque at the end one letter at a time
print(d)
d.extendleft('pqr')  # Extend the deque at the beginning one letter at a time
print(d)
print(d[9])
print(d.pop(), d.popleft())  # Pop from end, beginning
print(d)
print('-' * 60)


# namedtuple
President = namedtuple('President', 'first_name, last_name, party')  # Create named tuple
with specified fields
p = President('Theodore', 'Roosevelt', 'Republican')  # Create instance of named tuple
print(p, len(p))
print(p[0], p[1], p[-1])
print(p.first_name, p.last_name, p.party)  # Access tuple fields by name

p = President(last_name='Lincoln', party='Republican', first_name='Abraham')
print(p)
print(p.first_name, p.last_name)
print('-' * 60)
```

stdlib_containers.py

```
[('s', 19030), ('c', 16277), ('p', 14600), ('a', 10485), ('r', 10199), ('d', 10189),
('m', 9689), ('b', 9325), ('t', 8782), ('e', 7086)]
------------------------------------------------------------
a ['apricot', 'apple']
b ['banana', 'blueberry']
c ['cherry']
d ['date']
e ['elderberry']
f ['fig']
g ['guava', 'grape']
k ['kiwi']
l ['lemon', 'lime', 'lychee']
o ['orange']
p ['pomegranate', 'papaya', 'pear', 'persimmon', 'peach']
t ['tamarind']
w ['watermelon']
------------------------------------------------------------
deque(['a', 'b', 'c', 'd', 'e', 'f'])
deque(['l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f'])
deque(['l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f', 'm', 'n', 'o'])
deque(['r', 'q', 'p', 'l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f', 'm',
'n', 'o'])
a
o r
deque(['q', 'p', 'l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f', 'm', 'n'])
------------------------------------------------------------
President(first_name='Theodore', last_name='Roosevelt', party='Republican') 3
Theodore Roosevelt Republican
Theodore Roosevelt Republican
President(first_name='Abraham', last_name='Lincoln', party='Republican')
Abraham Lincoln
------------------------------------------------------------
```

The array module

- Efficient numeric arrays (fast than list)
- More compact than list
- Traditional arrays of the same type

The `array` module provides an array object which implements an efficient numeric array where each element is the same type. There are a number of different numeric types that can be used. The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the itemsize attribute.

This is efficient when you have a large number of numeric values to store, as it can take much less space than a normal list object, and is faster.

The numeric type flags are nearly the same as those use by the `struct` module.

The **ndarray** type in the **numpy** framework is very fast and efficient, and is typically used in science and engineering.

Example

array_examples.py

```
from sys import getsizeof
from array import array
from random import randint

values = [randint(1, 30000) for i in range(1000)]  # Create 1000 random values

print(f'Size of integer list: {getsizeof(values)}\n')

for datatype in 'i', 'h', 'L', 'Q', 'd':
    data_array = array(datatype, values)  # Create array object from values with various
datatypes
    print(f'Size of {datatype} array: {getsizeof(data_array)}  Contents:',
          data_array[:5], '...')  # Print size of array (will vary based on datatype)
    print()
```

array_examples.py

```
Size of integer list: 8856

Size of i array: 4064  Contents: array('i', [8108, 9854, 3176, 2352, 13904]) ...

Size of h array: 2064  Contents: array('h', [8108, 9854, 3176, 2352, 13904]) ...

Size of L array: 8064  Contents: array('L', [8108, 9854, 3176, 2352, 13904]) ...

Size of Q array: 8064  Contents: array('Q', [8108, 9854, 3176, 2352, 13904]) ...

Size of d array: 8064  Contents: array('d', [8108.0, 9854.0, 3176.0, 2352.0, 13904.0])
...
```

Table 14. array object type codes

Format	C Type	Python Type	Standard size	Notes
b	signed char	`int`	1	(1),(3)
B	unsigned char	`int`	1	(3)
h	short	`int`	2	(3)
H	unsigned short	`int`	2	(3)
i	int	`int`	4	(3)
I	unsigned int	`int`	4	(3)
l	long	`int`	4	(3)
L	unsigned long	`int`	4	(3)
q	long long	`int`	8	(2),(3)
Q	unsigned long long	`int`	8	(2), (3)
n	ssize_t	`int`		(4)
N	size_t	`int`		(4)
f	float	`float`	4	(5)
d	double	`float`	8	(5)

NOTE The 'q' and 'Q' type codes are available only if the platform C compiler used to build Python supports C long long, or, on Windows, __int64.

Emulating builtin types

- Inherit from builtin type
- Override special methods as needed
- Use super() to invoke base class's special methods

To emulate builtin types, you can inherit from builtin classes and override special methods as needed to get the desired behavior; other special methods will work in the normal way.

When overriding the special methods, you usually want to invoke the base class's special methods. Use the super() builtin function; the syntax is:

```
super().{dunder}specialmethod{dunder}(...)
```

To start from scratch, you can inherit from abstract base classes defined in the **collections.abc** module. These can be used as mixins, to make sure you're implemented the necessary methods for various container types. In other words, Sized plus Iterable will create a list-like object.

Example

container_abc.py

```python
from collections.abc import Sized, Iterator  # Abstract base classes, used similarly to
interfaces in Java or C#


class BadContainer(Sized):  # This class may not be instantiated without defining `len()`
    pass


class GoodContainer(Sized):
    def __len__(self):   # This class is fine, since `Sized` requires `len()` to be
implemented
        return 42


try:
    bad = BadContainer()  # Instantiating `BadContainer` raises an error.
except TypeError as err:
    print(err)
else:
    print(bad)

print()

try:
    good = GoodContainer()  # Instantiating `GoodContainer` is fine
except TypeError as err:
    print(err)
else:
    print(good)
    print(len(good))  # Builtin function `len()` works with all objects that inherit from
`Sized` (due to implementation of `len())`

print()


class MyIterator(Iterator):  # ABC `Iterator` provides abstract method `next`
    data = 'a', 'b', 'c'
    index = 0

    def __next__(self):  # Must be implemented for Iterators
        if self.index >= len(self.data):
            raise StopIteration
        else:
            return_val = self.data[self.index]
            self.index += 1
            return return_val


m = MyIterator()  # Create instance of `MyIterator`
for i in m:  # Iterate over the iterator instance
    print(i)
print()

print(hasattr(m, '__iter__'))  # Check to see if `m` is iterable
```

container_abc.py

```
Can't instantiate abstract class BadContainer with abstract method __len__

<__main__.GoodContainer object at 0x7f9708066bb0>
42

a
b
c

True
```

Table 15. Special Methods and Variables

Method or Variables	Description
`__new__(cls,…)`	Returns new object instance; Called before `__init__()`
`__init__(self,…)`	Object initializer (constructor)
`__del__(self)`	Called when object is about to be destroyed
`__repr__(self)`	Called by `repr()` builtin
`__str__(self)`	Called by `str()` builtin

`__eq__(self, other)`
`__ne__(self, other)`
`__gt__(self, other)`
`__lt__(self, other)`
`__ge__(self, other)`
`__le__(self, other)`
	Implement comparison operators ==, !=, >, <, >=, and ⇐. self is object on the left.

`__cmp__(self, other)`	Called by comparison operators if `__eq__`, etc., are not defined
`__hash__(self)`	Called by hash`()` builtin, also used by `dict`, `set`, and `frozenset` operations
`__bool__(self)`	Called by `bool()` builtin. Implements truth value (boolean) testing. If not present, `bool()` uses `len()`
`__unicode__(self)`	Called by `unicode()` builtin

`__getattr__(self, name)`
`__setattr__(self, name, value)`
`__delattr__(self, name)`
	Override normal fetch, store, and deleter

`__getattribute__(self, name)`	Implement attribute access for new-style classes
`__get__(self, instance)`	__set__(self, instance, value)
`__del__(self, instance)`	Implement descriptors
`__slots__ = variable-list`	Allocate space for a fixed number of attributes.
`__metaclass__ = callable`	Called instead of `type()` when class is created.
`__instancecheck__(self, instance)`	Return `True` if instance is an instance of class
`__subclasscheck__(self, instance)`	Return `True` if instance is a subclass of class
`__call__(self, …)`	Called when instance is called as a function.
`__len__(self)`	Called by `len()` builtin
`__getitem__(self, key)`	Implements `self[key]`
`__setitem__(self, key, value)`	Implements `self[key] = value`
`__selitem__(self, key)`	Implements `del self[key]`
`__iter__(self)`	Called when iterator is applied to container

Method or Variables	Description
`__reversed__(self)`	Called by `reversed()` builtin
`__contains__(self, object)`	Implements `in` operator

`__add__(self, other)`
`__sub__(self, other)`
`__mul__(self, other)`
`__floordiv__(self, other)`
`__mod__(self, other)`
`__divmod__(self, other)`
`__pow__(self, other[, modulo])`
`__lshift__(self, other)`
`__rshift__(self, other)`
`__and__(self, other)`
`__xor__(self, other)`
`__or__(self, other)`
	Implement binary arithmetic operators `+`, `-, `*, /, //, %, `**`, `<<`, `>>`, `&`, `^`, and `|`. **self** is object on left side of expression.

`__div__(self,other)`
`__truediv__(self,other)`
	Implement binary division operator `/`. `__truediv__()` is called if `__future__.division` is in effect.

`__radd__(self, other)`
`__rsub__(self, other)`
`__rmul__(self, other)`
`__rdiv__(self, other)`
`__rtruediv__(self, other)`
`__rfloordiv__(self, other)`
`__rmod__(self, other)`
`__rdivmod__(self, other)`
`__rpow__(self, other)`
`__rlshift__(self, other)`
`__rrshift__(self, other)`
`__rand__(self, other)`
`__rxor__(self, other)`
`__ror__(self, other)`
	Implement binary arithmetic operators with swapped operands. (Used if left operand does not support the corresponding operation)

`__iadd__(self, other)`
`__isub__(self, other)`
`__imul__(self, other)`
`__idiv__(self, other)`
`__itruediv__(self, other)`
`__ifloordiv__(self, other)`
`__imod__(self, other)`
`__ipow__(self, other[, modulo])`
`__ilshift__(self, other)`
`__irshift__(self, other)`
`__iand__(self, other)`
`__ixor__(self, other)`
`__ior__(self, other)`
	Implement augmented (+=, -=, etc.) arithmetic operators

`__neg__(self)`
`__pos__(self)`
`__abs__(self)`
`__invert__(self)`
	Implement unary arithmetic operators -, +, abs(), and ~

Method or Variables	Description

`__oct__(self)`
`__hex__(self)`
	Implement `oct()` and `hex()` builtins

`__index__(self)`	Implement `operator.index()`
`__coerce__(self, other)`	Implement "mixed-mode" numeric arithmetic.

Creating list-like containers

- Inherit from list
- Override special methods as needed
- Commonly overridden methods
  - __getitem__
  - __setitem__
  - __append__

To create a list-like container, inherit from list. Commonly overridden methods include __getitem__ and __setitem__.

Example

multiindexlist.py

```
class MultiIndexList(list):  # Define new class that inherits from list

    def __getitem__(self, item):  # Redefine __getitem__ which implements []
        if isinstance(item, tuple):  # Check to see if index is tuple
            if len(item) == 0:
                raise ValueError("Tuple must be non-empty")
            else:
                tmp_list = []
                for index in item:
                    tmp_list.append(
                        super().__getitem__(index)  # Call list.__getitem__() for each
index in tuple
                    )
                return tmp_list
        else:
            return super().__getitem__(item)  # Call the normal __getitem__()


if __name__ == '__main__':
    m = MultiIndexList(
        'banana peach nectarine fig kiwi lemon lime'.split()
    )  # Initialize a MultiIndexList
    m.append('apple')  # Add an element (works like normal list)
    m.append('mango')
    print(m)

    print(m[0])
    print(m[1])
    print(m[5, 2, 0])  # Index with tuple
    print(m[:4])
    print(len(m))
    print(m[5, ])
    print(m[:2, -2:])
    print()
    print(m)
    m.extend(['durian', 'kumquat'])
    print(m)
    print()
    for fruit in m:
        print(fruit)
    print(len(fruit))
```

multiindexlist.py

```
['banana', 'peach', 'nectarine', 'fig', 'kiwi', 'lemon', 'lime', 'apple', 'mango']
banana
peach
['lemon', 'nectarine', 'banana']
['banana', 'peach', 'nectarine', 'fig']
9
['lemon']
[['banana', 'peach'], ['apple', 'mango']]

['banana', 'peach', 'nectarine', 'fig', 'kiwi', 'lemon', 'lime', 'apple', 'mango']
['banana', 'peach', 'nectarine', 'fig', 'kiwi', 'lemon', 'lime', 'apple', 'mango',
'durian', 'kumquat']

banana
peach
nectarine
fig
kiwi
lemon
lime
apple
mango
durian
kumquat
7
```

Creating dict-like containers

- Inherit from dict
- Implement special methods
- Commonly overridden methods
    - __getitem__
    - __haskey__
    - __setitem__

To create custom dictionaries, inherit from dict and implement special methods as needed. Commonly overridden special methods include __getitem__(), __setitem__(), and __haskey__().

Example

stringkeydict.py

```
class StringKeyDict(dict):  # Create class that inherits from dict
    def __setitem__(self, key, value):  # Overwrite how values are stored in the dict via
_DICT_[_KEY_] = _VALUE_
        if isinstance(key, str):   # Make sure key is a string
            super().__setitem__(key, value)  # Use dict's setitem to set value if it is
not a key
        else:
            raise TypeError("Keys must be strings not {}s".format(  # Raise error if non-
string key is used
                type(key).__name__
            ))


if __name__ == '__main__':
    d = StringKeyDict(a=10, b=20)   # Create and initialize StringKeyDict instance
    for k, v in [('c', 30), ('d', 40), (1, 50), (('a', 1), 60), (5.6, 201)]:
        try:
            print("Setting {} to {}".format(k, v), end=' ')
            d[k] = v   # Try to add various key/value pairs
        except TypeError as err:
            print(err)  # Error raised on non-string key
        else:
            print('SUCCESS')

    print()
    print(d)
```

stringkeydict.py

```
Setting c to 30 SUCCESS
Setting d to 40 SUCCESS
Setting 1 to 50 Keys must be strings not ints
Setting ('a', 1) to 60 Keys must be strings not tuples
Setting 5.6 to 201 Keys must be strings not floats

{'a': 10, 'b': 20, 'c': 30, 'd': 40}
```

Free-form containers

- Easy to create hybrid containers
- Objects may implement any special methods
- Create list+dict, ordered set
- Be creative

There's really no limit to the types of objects you can create. You can make hybrids that act like lists and dictionaries at the same time. Just implement the special methods required for this behavior.

A well-known example of this is the **Element** class in the lxml.etree module. An Element is a list of its children, and at the same time is a dictionary of its XML attributes:

```
e = Element(...)
child_element = e[0]
attr_value = e.get("attribute")
```

Chapter 11 Exercises

Exercise 11-1 (maxlist.py)

Create a new type, MaxList, that will only grow to a certain size. It should raise an IndexError if the user attempts to add an item beyond the limit.

HINT: you will need to override the append() and extend() methods; to be thorough, you would need to override __init__() as well, so that the initializer doesn't have more than the maximum number of items., and pop(), and maybe some others.

For the ambitious (ringlist.py): Modify MaxList so that once it reaches maximum size, appending to the list also removes the first element, so the list stays constant size.

Exercise 11-2 (strdict.py)

Create a new type, NormalStringDict, that only allows strings as keys *and* values. Values are normalized by making them lower case and remove all whitespace.

Chapter 12: Generators and Other Iterables

Objectives

- Unpack function arguments
- Unpack iterables with wildcards
- Use lambda functions for brevity
- Understand Python iterables
- Select and transform data with list comprehensions
- Create generators in 3 different ways
- Implement data pipelines with coroutines

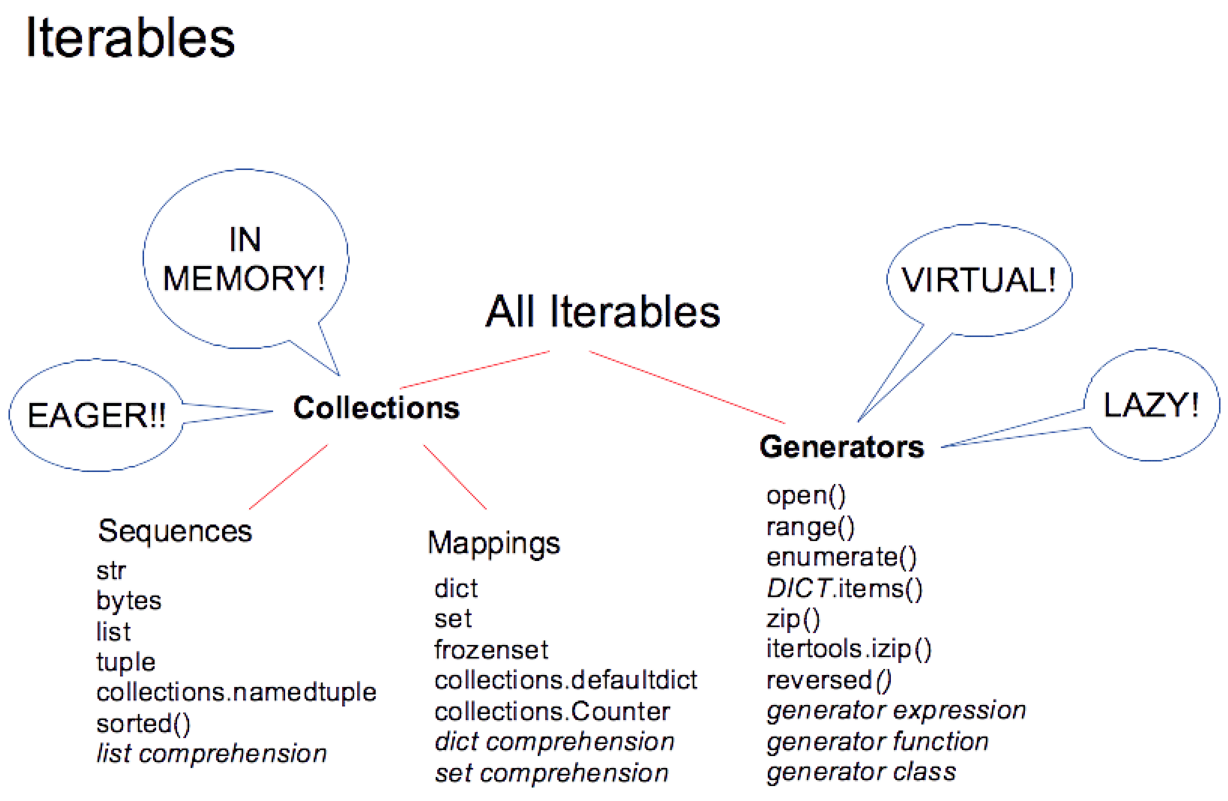
Iterables

- An iterable is an expression that can be looped through with for
- Some iterables are collections (list, tuple, str, bytes, dict, set)
- Many iterables are generators(enumerate(), dict.items(), open(), zip(), reversed(), etc)

for is one of the most powerful operators in Python. It it used for looping through a set of values. The set of values is provided by an iterable. Python has many builtin iterables – a file object, for instance, which allows iterating through the lines in a file.

All collections (list, tuple, str, bytes) are iterables. They keep all their values in memory.

A generator is an iterable that does not keep all its values in memory – it creates them one at a time as needed, and feeds them to the for loop. This is a **Good Thing**, because it saves memory.



Unpacking function arguments

- Convert from iterable to list of items
- Use * or **

What do you do if you have a list (or other iterable) of three values, and you want to pass them to a method that expects three positional arguments? You could say value[0], value[1], value[2], etc., but there's a more Pythonic way:

Use * to unpack the iterable into individual items. The iterable must have the same number of values as the number of parameters in the function. However, you can combine individual arguments with unpacking:

```
foo(5, 10, *values)
```

In a similar way, use two asterisks to unpack a dictionary.

Example

param_unpacking.py

```python
from datetime import date

dates = [
    (1968, 10, 11),
    (1968, 12, 21),
    (1969, 3, 3),
    (1969, 5, 18),
    (1969, 7, 16),
    (1969, 11, 14),
    (1970, 4, 11),
    (1971, 1, 31),
    (1971, 7, 26),
    (1972, 4, 16),
    (1927, 12, 7),
]  # tuple of dates

for dt in dates:
    d = date(*dt)  # instead of date(dt[0], dt[1], dt[2])
    print(d)

print()

fruits = ["pomegranate", "cherry", "apricot", "date", "Apple", "lemon", "Kiwi",
          "ORANGE", "lime", "Watermelon", "guava", "papaya", "FIG", "pear",
          "banana", "Tamarind", "persimmon", "elderberry", "peach", "BLUEberry",
          "lychee", "grape"]

sort_opts = {
    'key': lambda e: e.lower(),
    'reverse': True,
}  # config info in dictionary

sorted_fruits = sorted(fruits, **sort_opts)  # dictionary converted to named parameters
print(sorted_fruits)
```

param_unpacking.py

```
1968-10-11
1968-12-21
1969-03-03
1969-05-18
1969-07-16
1969-11-14
1970-04-11
1971-01-31
1971-07-26
1972-04-16
1927-12-07

['Watermelon', 'Tamarind', 'pomegranate', 'persimmon', 'pear', 'peach', 'papaya',
'ORANGE', 'lychee', 'lime', 'lemon', 'Kiwi', 'guava', 'grape', 'FIG', 'elderberry',
'date', 'cherry', 'BLUEberry', 'banana', 'apricot', 'Apple']
```

Iterable unpacking

- Frequently used with `for` loops
- Can be used anywhere
- Commonly used with `enumerate()` and `DICT.items()`

It is convenient to unpack an iterable into a list of variables. It can be done with any iterable, and is frequently done with the loop variables of a `for` loop, rather than unpacking the value of each iteration separately.

```
var1, … = iterable
```

TIP Underscore (`_`) can be used as a variable name for values you don't care about.

Example

iterable_unpacking.py

```
values = ['a', 'b', 'c']

x, y, z = values  # unpack values (which is an iterable) into individual variables

print(x, y, z)
print()

people = [
    ('Bill', 'Gates', 'Microsoft'),
    ('Steve', 'Jobs', 'Apple'),
    ('Paul', 'Allen', 'Microsoft'),
    ('Larry', 'Ellison', 'Oracle'),
    ('Mark', 'Zuckerberg', 'Facebook'),
    ('Sergey', 'Brin', 'Google'),
    ('Larry', 'Page', 'Google'),
    ('Linux', 'Torvalds', 'Linux'),
]

for row in people:
    first_name, last_name, _ = row  # unpack row into variables
    print(first_name, last_name)
print()

for first_name, last_name, _ in people:  # a for loop unpacks if there is more than one
variable
    print(first_name, last_name)
print()

# extended unpacking
values = ['a', 'b', 'c', 'd', 'e', 'f']
x, y, *z = values
print(x, y, z)

x, *y, z = values
print(x, y, z)

*x, y, z = values
print(x, y, z)
```

iterable_unpacking.py

```
a b c

Bill Gates
Steve Jobs
Paul Allen
Larry Ellison
Mark Zuckerberg
Sergey Brin
Larry Page
Linux Torvalds

Bill Gates
Steve Jobs
Paul Allen
Larry Ellison
Mark Zuckerberg
Sergey Brin
Larry Page
Linux Torvalds

a b ['c', 'd', 'e', 'f']
a ['b', 'c', 'd', 'e'] f
['a', 'b', 'c', 'd'] e f
```

Extended iterable unpacking

- Allows for one "wild card"
- Allows common "first, rest" unpacking

When unpacking iterables, sometimes you want to grab parts of the iterable as a group. This is provided by extended iterable unpacking.

One (and only one) variable in the result of unpacking can have a star prepended. This variable will receive all values from the iterable that do not go to other variables.

NOTE Extended variable unpacking is not available in Python 2.

Example

extended_iterable_unpacking.py

```
values = ['a', 'b', 'c', 'd', 'e']  # values has 6 elements

x, y, *z = values  # {splat} takes all extra elements from iterable
print("x: {}    y: {}    z: {}".format(x, y, z))
print()

x, *y, z = values  # {splat} takes all extra elements from iterable
print("x: {}    y: {}    z: {}".format(x, y, z))
print()

*x, y, z = values  # {splat} takes all extra elements from iterable
print("x: {}    y: {}    z: {}".format(x, y, z))
print()

people = [
    ('Bill', 'Gates', 'Microsoft'),
    ('Steve', 'Jobs', 'Apple'),
    ('Paul', 'Allen', 'Microsoft'),
    ('Larry', 'Ellison', 'Oracle'),
    ('Mark', 'Zuckerberg', 'Facebook'),
    ('Sergey', 'Brin', 'Google'),
    ('Larry', 'Page', 'Google'),
    ('Linux', 'Torvalds', 'Linux'),
]

for *name, _ in people:  # name gets all but the last field
    print(name)
print()
```

extended_iterable_unpacking.py

```
x: a    y: b    z: ['c', 'd', 'e']

x: a    y: ['b', 'c', 'd']    z: e

x: ['a', 'b', 'c']    y: d    z: e

['Bill', 'Gates']
['Steve', 'Jobs']
['Paul', 'Allen']
['Larry', 'Ellison']
['Mark', 'Zuckerberg']
['Sergey', 'Brin']
['Larry', 'Page']
['Linux', 'Torvalds']
```

What exactly is an iterable?

- Object that provides an *iterator*
- Can be **collection** or **generator**

An *iterable* is an object that provides an *iterator* (via the special method __iter__. An iterator is an object that responds to the **next()** builtin function, via the special method __next__(). In other words, an iterator is an object which can be looped over with a **for** loop.

All generators are iterables. Most sequence and mapping types are also iterables. Generators are also iterators; **next()** can be used on them directly.

For some iterables (including most collections), you can not use `next()` on them directly; use the builtin function `iter()` to get the iterator, then use `next()` on the result.

```
>>> r = range(1, 4)
>>> i = iter(r)
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

NOTE A `for` loop is really a `while` loop in disguise. It repeatedly calls `next()` on an iterator, and stops when `StopIteration` is raised.

List comprehensions

- Shortcut syntax for a `for` loop
- Selects and/or modifies values
- Can be faster

A *list comprehension* is a Python idiom that creates a shortcut for a `for` loop. It returns a copy of a list with every element transformed via an expression. Functional programmers refer to this as a mapping function.

A loop like this:

```
results = []
for var in sequence:
    results.append(expr)   # where expr involves var
```

can be rewritten as

```
results = [ expr for var in sequence ]
```

A conditional `if` may be added:

```
results = [ expr for var in sequence if expr ]
```

A list comprehension can both select and modify values from an iterable and append them to a new list.

Common uses:

- convert values to strings
- pull fields out of a nested data structure
- select items and make them lower case
- normalize strings by cleaning up whitespace

Example

list_comprehensions.py

```python
fruits = ['watermelon', 'apple', 'mango', 'kiwi', 'apricot', 'lemon', 'guava']

ufruits = [fruit.upper() for fruit in fruits]        # Simple transformation of all elements
afruits = [fruit.title() for fruit in fruits if fruit.startswith('a')]  # Transformation of selected elements only

print("ufruits:", ufruits)
print("afruits:", afruits)
print()

values = [2, 42, 18, 39.7, 92, '14', "boom", ['a', 'b', 'c']]

doubles = [v * 2 for v in values]   # Any kind of data is OK

print("doubles:", doubles, '\n')

nums = [x for x in values if isinstance(x, int)]   # Select only integers from list
print(nums, '\n')

dirty_strings = ['   Gronk    ', 'PULABA       ', '        floog']

clean = [d.strip().lower() for d in dirty_strings]
for c in clean:
    print(">{}<".format(c), end=' ')
print("\n")

suits = 'Clubs', 'Diamonds', 'Hearts', 'Spades'
ranks = '2 3 4 5 6 7 8 9 10 J Q K A'.split()

deck = [(rank, suit) for suit in suits for rank in ranks]   # More than one for is OK

for rank, suit in deck:
    print("{}-{}".format(rank, suit))
```

list_comprehensions.py

```
ufruits: ['WATERMELON', 'APPLE', 'MANGO', 'KIWI', 'APRICOT', 'LEMON', 'GUAVA']
afruits: ['Apple', 'Apricot']

doubles: [4, 84, 36, 79.4, 184, '1414', 'boomboom', ['a', 'b', 'c', 'a', 'b', 'c']]

[2, 42, 18, 92]

>gronk< >pulaba< >floog<

2-Clubs
3-Clubs
4-Clubs
5-Clubs
6-Clubs
7-Clubs
```

...

Generators

- Lazy iterables
- Do not contain data
- Provide values on demand
- Save memory

As part of the transition from Python 2 to Python 3, many operations that returned lists were changed to return **generators**

A generator is an object that provides values on demand (AKA "lazy"), rather than storing all the values (AKA "eager"). Generators are usually based on some other iterable. Another way of saying this is that a generator returns an iterator that returns a new value each time next() is called on it, until there are no more values.

The big advantage of generators is saving memory. They act as a *view* over a set of data.

Generators may only be used once. After the last value is provided, the generator must be recreated in order to start over.

Generators may not be indexed, and have no length. The only thing to do with a generator is to loop over it with **for**.

There are three ways to create generators in Python:

- generator expression
- generator function
- generator class

Generator Expressions

- Like list comprehensions, but create a generator object
- Use parentheses rather than brackets
- Saves memory

A generator expression is similar to a list comprehension, but it provides a generator instead of a list. That is, while a list comprehension returns a complete list, the generator expression returns one item at a time. The generator does not contain a copy of the source iterable, so it saves memory. In many cases generators are also faster than list comprehensions.

The main difference in syntax is that the generator expression uses parentheses rather than brackets.

If a generator expression is passed as the only argument to a function, the extra parentheses are not needed.

```
my_func(float(i) for i in values)
```

Generator expressions are especially useful with functions like sum(), min(), and max() that reduce an iterable input to a single value.

NOTE There is an implied yield statement at the beginning of the expression.

Example

generator_expressions.py

```python
fruits = ['watermelon', 'apple', 'mango', 'kiwi', 'apricot', 'lemon', 'guava']

ufruits = (fruit.upper() for fruit in fruits)  # These are all exactly like the list comprehension example, but return generators rather than lists
afruits = (fruit.title() for fruit in fruits if fruit.startswith('a'))

print("ufruits:", " ".join(ufruits))
print("afruits:", " ".join(afruits))
print()

values = [2, 42, 18, 92, "boom", ['a', 'b', 'c']]
doubles = (v * 2 for v in values)

print("doubles:", end=' ')
for d in doubles:
    print(d, end=' ')
print("\n")

nums = (int(s) for s in values if isinstance(s, int))
for n in nums:
    print(n, end=' ')
print("\n")

dirty_strings = ['   Gronk    ', 'PULABA       ', '        floog']

clean = (d.strip().lower() for d in dirty_strings)
for c in clean:
    print(">{}<".format(c), end=' ')
print("\n")

powers = ((i, i ** 2, i ** 3) for i in range(1, 11))
for num, square, cube in powers:
    print("{:2d} {:3d} {:4d}".format(num, square, cube))
print()
```

generator_expressions.py

```
ufruits: WATERMELON APPLE MANGO KIWI APRICOT LEMON GUAVA
afruits: Apple Apricot

doubles: 4 84 36 184 boomboom ['a', 'b', 'c', 'a', 'b', 'c']

2 42 18 92

>gronk< >pulaba< >floog<

 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Generator functions

- Mostly like a normal function
- Use yield rather than return
- Maintains state

A generator is like a normal function, but instead of a return statement, it has a yield statement. Each time the yield statement is reached, it provides the next value in the sequence. When there are no more values, the function calls return, and the loop stops. A generator function maintains state between calls, unlike a normal function.

Example

sieve_generator.py

```python
def next_prime(limit):
    flags = set()  # initialize empty set (to be used for "is-prime" flags

    for i in range(2, limit):
        if i in flags:
            continue
        for j in range(2 * i, limit + 1, i):
            flags.add(j)  # add non-prime elements to set
        yield i  # execution stops here until next value is requested by for-in loop


np = next_prime(200)  # next_prime() returns a generator object
for prime in np:  # iterate over yielded primes
    print(prime, end=' ')
```

sieve_generator.py

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109
113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199
```

Example

line_trimmer.py

```
def trimmed(file_name):
    with open(file_name) as file_in:
        for line in file_in:
            yield line.rstrip('\n\r')  # 'yield' causes this function to return a
generator object

mary_in = trimmed('../DATA/mary.txt')
for trimmed_line in mary_in:
    print(trimmed_line)
```

line_trimmer.py

```
Mary had a little lamb,
Its fleece was white as snow,
And everywhere that Mary went
The lamb was sure to go
```

Coroutines

- **yield** is an expression (can be assigned to)
- uses ***generator*.send()**
- yield returns None by default

When writing generator functions, you can send a value *back* to the generator. This is accomplished by calling *generator*.send(), and in the generator it becomes the return value of the **yield** statement. This makes the generator into a *coroutine*. While generators and coroutines are similar, they are not the same thing. Generators tend to be *producers*, while coroutines tend to be *consumers*.

When assigning to the **yield** statement, you need to call next() once to "prime" the coroutine and poise it to receive the first input, and thus send the first output. There's an "off-by-one" feeling to this.

Coroutines are not designed for iteration. They can be used to set up pipelines (sequences of coroutines that each do one interesting thing to your data. **send()** puts data into the pipeline.

This can be useful for async-like coding.

```
...
   ham = yield spam
...


...
   next(p)
   p.send('foo')
   print(next(p))
...
```

See http://dabeaz.com/coroutines/ for an in-depth look at coroutines.

Example

coroutine_example.py

```python
def coroutine():  # Define a coroutine function
    in_value = ''
    while True:
        in_value = yield in_value.upper()  # Yield gets the result of send() AND provides
the next value
        print('in_value:', in_value)


c = coroutine()  # Create instance of coroutine

print(c)  # c is a coroutine object (also a generator)
print(next(c))  # next() will get the next value and prime the coroutine. You _must_ call
next before calling send()
print()

for letter in "alpha", "beta", 'gamma':
    print("out_value:", c.send(letter))  # Send a value to the coroutine
print()


def faux_range():  # define a generator
    i = 0
    while i < 4:
        yield i
        i += 1


def spam():
    yield from faux_range()  # Define generator; yield from _delegates_ to another
generator


s = spam()  # Create instance of generator

for x in s:  # Looping through a spam() instance effictively loops over faux_range()
    print(x)
```

coroutine_example.py

```
<generator object coroutine at 0x7f906806c900>


in_value: alpha
out_value: ALPHA
in_value: beta
out_value: BETA
in_value: gamma
out_value: GAMMA

0
1
2
3
```

Generator classes

- Implement __iter__ and __next__
- Emit values via **for** or **next()**
- Raise StopIteration exception when there are no more values to emit

The most flexible way to create a generator is by defining a generator class.

Such a class must implement two special methods:

__iter__ must return an object that implements __next__. In most cases, this is the same class, so __iter__ just returns **self**.

__next__ returns the next value from the generator. This can be in a loop, or in sequential statements, or any combination.

When there are no more values to return, __next__ should raise **StopIteration**.

Example

trimmedfile.py

```
class TrimmedFile:
    def __init__(self, file_name):  # constructor is passed file name
        self._file_in = open(file_name)

    def __iter__(self):  # A generator must implement iter(), which must return an
iterator. Typically it returns self, as the generator _is_ the iterator
        return self

    def __next__(self):  # next() returns the next value of the generator
        line = self._file_in.readline()
        if line == '':
            raise StopIteration  # Raise StopIteration when there are no more values to
generate
        else:
            return line.rstrip('\n\r')  # The actual work of this generator -- remove the
newline from the line


if __name__ == '__main__':
    trimmed = TrimmedFile('../DATA/mary.txt')  # To use the generator, create an instance
and iterator over it.
    for line in trimmed:
        print(line)
```

trimmedfile.py

```
Mary had a little lamb,
Its fleece was white as snow,
And everywhere that Mary went
The lamb was sure to go
```

Chapter 12 Exercises

Exercise 12-1 (pres_upper.py)

1. Use a list comprehension to read all the presidents' first and last names into a list of tuples
2. Use another list comprehension to make a new list with the names joined by spaces, and in upper case.
3. Loop through the upper case list and print out the names one per line.

Exercise 12-2 (pres_upper_gen.py)

Redo pres_upper.py but use two generator expressions rather than two list comprehensions.

Exercise 12-3 (pres_gen.py)

Write a generator function to provide a sequence of the names of presidents (in "FIRSTNAME LASTNAME" format) from the presidents.txt file. They should be provided in the same order they are in the file. You should not read the entire file into memory, but one-at-a-time from the file.

Exercise 12-4 (fauxrange.py)

Write a generator class named FauxRange that emulates the builtin range() function. Instances should take up to three arguments, and provide a range of integers (or, consider using floats — does that change the class?).

Chapter 13: Metaprogramming

Objectives

- Learn what metaprogramming means
- Access local and global variables by name
- Inspect the details of any object
- Use attribute functions to manipulate an object
- Design decorators for classes and functions
- Define classes with the type() function
- Create metaclasses

Metaprogramming

- Writing code that writes (or at least modifies) code
- Can simplify some kinds of programs
- Not as hard as you think!
- Considered deep magic in other languages

Metaprogramming is writing code that generates or modifies other code. It includes fetching, changing, or deleting attributes, and writing functions that return functions (AKA factories).

Metaprogramming is easier in Python than many other languages. Python provides explicit access to objects, even the parts that are hidden or restricted in other languages.

For instance, you can easily replace one method with another in a Python class, or even in an object instance. In Java, this would be deep magic requiring many lines of code.

globals() and locals()

- Contain all variables in a namespace
- globals() returns all global objects
- locals() returns all local variables

The **globals()** builtin function returns a dictionary of all global objects. The keys are the object names, and the values are the objects values. The dictionary is "live" — changes to the dictionary affect global variables.

The **locals()** builtin returns a dictionary of all objects in local scope.

Example

globals_locals.py

```python
from pprint import pprint  # import prettyprint function

spam = 42  # global variable
ham = 'Smithfield'


def eggs(fruit):  # function parameters are local
    name = 'Lancelot'  # local variable
    idiom = 'swashbuckling'  # local variable
    print("Globals:")
    pprint(globals())  # globals() returns dict of all globals
    print()
    print("Locals:")
    pprint(locals())  # locals() returns dict of all locals


eggs('mango')
```

globals_locals.py

```
Globals:
{'__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>,
 '__cached__': None,
 '__doc__': None,
 '__file__': '/Users/jstrick/curr/courses/python/common/examples/globals_locals.py',
 '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x7fa5d008ccd0>,
 '__name__': '__main__',
 '__package__': None,
 '__spec__': None,
 'eggs': <function eggs at 0x7fa5d00e8040>,
 'ham': 'Smithfield',
 'pprint': <function pprint at 0x7fa5c00de550>,
 'spam': 42}

Locals:
{'fruit': 'mango', 'idiom': 'swashbuckling', 'name': 'Lancelot'}
```

The inspect module

- Simplifies access to metadata
- Provides user-friendly functions for testing metadata

The `inspect` module provides user-friendly functions for accessing Python metadata.

Example

inspect_ex.py

```
import inspect
import geometry
from carddeck import CardDeck

deck = CardDeck("Leonard")

things = (
    geometry,
    geometry.circle_area,
    CardDeck,
    CardDeck.get_ranks,
    deck,
    deck.shuffle,
)

print("Name               Module?  Function?  Class?  Method?")
for thing in things:
    try:
        thing_name = thing.__name__
    except AttributeError:
        thing_name = type(thing).__name__ + " instance"
    print("{:18s} {!s:6s}   {!s:6s}     {!s:6s}  {!s:6s}".format(
        thing_name,
        inspect.ismodule(thing),  # test for module
        inspect.isfunction(thing),  # test for function
        inspect.isclass(thing),  # test for class
        inspect.ismethod(thing),
    ))


print()
def spam(p1, p2='a', *p3, p4, p5='b', **p6):  # define a function
    print(p1, p2, p3, p4, p5, p6)

# get argument specifications for a function
print("Function spec for Ham:", inspect.getfullargspec(spam))
print()

# get frame (function call stack) info
print("Current frame:", inspect.getframeinfo(inspect.currentframe()))
```

inspect_ex.py

```
Name              Module?  Function?  Class?  Method?
geometry          True     False      False   False
circle_area       False    True       False   False
CardDeck          False    False      True    False
get_ranks         False    False      False   True
CardDeck instance False    False      False   False
shuffle           False    False      False   True

Function spec for Ham: FullArgSpec(args=['p1', 'p2'], varargs='p3', varkw='p6',
defaults=('a',), kwonlyargs=['p4', 'p5'], kwonlydefaults={'p5': 'b'}, annotations={})

Current frame:
Traceback(filename='/Users/jstrick/curr/courses/python/common/examples/inspect_ex.py',
lineno=40, function='<module>', code_context=['print("Current frame:",
inspect.getframeinfo(inspect.currentframe()))\n'], index=0)
```

Table 16. inspect module convenience methods

Method(s)	Description
`ismodule()`, `isclass()`, `ismethod()`, `isfunction()`, `isgeneratorfunction()`, `isgenerator()`, `istraceback()`, `isframe()`, `iscode()`, `isbuiltin()`, `isroutine()`	check object types
`getmembers()`	get members of an object that satisfy a given condition
`getfile()`, `getsourcefile()`, `getsource()`	find an object’s source code
`getdoc()`, `getcomments()`	get documentation on an object
`getmodule()`	determine the module that an object came from
`getclasstree()`	arrange classes so as to represent their hierarchy
`getargspec()`, `getargvalues()`	get info about function arguments
`formatargspec()`, `formatargvalues()`	format an argument spec
`getouterframes()`, `getinnerframes()`	get info about frames
`currentframe()`	get the current stack frame
`stack()`, `trace()`	get info about frames on the stack or in a traceback

Working with attributes

- Objects are dictionaries of attributes
- Special functions can be used to access attributes
- Attributes specified as strings
- Syntax

```
getattr(object, attribute [,defaultvalue] )
hasattr(object, attribute)
setattr(object, attribute, value)
delattr(object, attribute)
```

All Python objects are essentially dictionaries of attributes. There are four special builtin functions for managing attributes. These may be used to programmatically access attributes when you have the name as a string.

getattr() returns the value of a specified attribute, or raises an error if the object does not have that attribute. `getattr(a,'spam')` is the same as `a.spam`. An optional third argument to getattr() provides a default value for nonexistent attributes (and does not raise an error).

hasattr() returns the value of a specified attribute, or None if the object does not have that attribute.

setattr() an attribute to a specified value.

delattr() deletes an attribute and its corresponding value.

Example

attributes.py

```python
class Spam():

    def eggs(self, msg):  # create attribute
        print("eggs!", msg)


s = Spam()

s.eggs("fried")

print("hasattr()", hasattr(s, 'eggs'))  # check whether attribute exists

e = getattr(s, 'eggs')  # retrieve attribute
e("scrambled")


def toast(self, msg):
    print("toast!", msg)


setattr(Spam, 'eggs', toast)  # set (or overwrite) attribute

s.eggs("buttered!")

delattr(Spam, 'eggs')  # remove attribute

try:
    s.eggs("shirred")
except AttributeError as err:  # missing attribute raises error
    print(err)
```

attributes.py

```
eggs! fried
hasattr() True
eggs! scrambled
toast! buttered!
'Spam' object has no attribute 'eggs'
```

Adding instance methods

- Use setattr()
- Add instance method to class
- Add instance method to instance

Using `setattr()`, it is easy to add instance methods to classes. Just add a function object to the class. Because it is part of the class itself, it will automatically be bound to the instance. Remember that an instance method expects `self` as the first parameter. In fact, this is the meaning of a bound instance — it is "bound" to the instance, and therefore when called, it is passed the instance as the first parameter.

Once added, the method may be called from any existing *or new* instance of the class.

To add an instance method to an *instance* takes a little more effort. Because it’s not being added to the class, it is not automatically bound. The function needs to know what instance it should be bound to. This can be accomplished with the `types.MethodType()` function.

Pass the function and the instance to `MethodType()`.

Example

adding_instance_methods.py

```python
from types import MethodType

class Dog(): # Define Dog type
    pass

d1 = Dog()  # Create instance of Dog

def bark(self):  # Define (unbound) function
    print("Woof! woof!")

setattr(Dog, "bark", bark)  # Add function to class (which binds it as an instance
method)

d2 = Dog() # Define another instance of Dog

d1.bark()  # New function can be called from either instance
d2.bark()

def wag(self): # Create another unbound function
    print("Wagging...")

setattr(d1, "wag", MethodType(wag, d1))  # Add function to instance after passing it
through MethodType()

d1.wag()  # Call instance method
try:
    d2.wag()  # Instance method not available - only bound to d1
except AttributeError as err:
    print(err)
```

adding_instance_methods.py

```
Woof! woof!
Woof! woof!
Wagging...
'Dog' object has no attribute 'wag'
```

Callable classes

- Convenient for one-method classes
- Really "callable instances"
- Implement __call__
- Convenient for one-method classes
- Useful for decorators

Any class instance may be made callable by implementing the special method `call`. This means that rather than saying:

```
sc = SomeClass()
sc.some_method()
```

you can say

```
sc = SomeClass()
sc()
```

What’s the advantage? Really, not too much. It just saves having to call a method from the instance, letting you call the instance itself. The use case is for classes that only have one method.

You can think of a callable class as a function that can also keep some state. As with many object-oriented features, its main purpose is to simplify the user interface.

One good use of callable classes is for implementing decorators as classes, rather than functions.

NOTE See `memorychecker.py` in the EXAMPLES folder for a real-life use of a callable class to make it easy to check the current memory footprint.

Example

callable_class.py

```
class TagWrapper():
    def __init__(self, tag):
        self._tag = tag

    def wrap(self, text):
        return '<{0}>{1}</{0}>'.format(self._tag, text)

class HTMLWrapper():

    def __init__(self, tag):
        self._tag = tag

    def __call__(self, text):  # Define function to be called when instance is called
        return '<{0}>{1}</{0}>'.format(self._tag, text)

if __name__ == '__main__':
    # non-callable class
    t = TagWrapper('h1')
    print(t.wrap('foo'))
    print(t.wrap('bar'))
    print()

    # callable class
    h1 = HTMLWrapper('h1')  # Create instance of "callable class"
    print(h1('spam'))  # Instance is callable -- essentially  h1.call('spam')
    div = HTMLWrapper('div')
    print(div('ham'))
    print(div('toast'))
    print(div('jam'))
```

callable_class.py

```
<h1>foo</h1>
<h1>bar</h1>

<h1>spam</h1>
<div>ham</div>
<div>toast</div>
<div>jam</div>
```

Decorators

- Classic design pattern
- Built into Python
- Implemented via functions or classes
- Can decorate functions or classes
- Can take parameters (but not required to)
- functools.wraps() preserves function’s properties

In Python, many decorators are provided by the standard library, such as property() or classmethod()

A decorator is a component that modifies some other component. The purpose is typically to add functionality, but there are no real restrictions on what a decorator can do. Many decorators register a component with some other component. For instance, the @app.route() decorator in Flask maps a URL to a view function.

As another example, **unittest** provides decorators to skip tests. A very common decorator is **@property**, which converts a class method into a property object.

A decorator can be any *callable*, which means it can be a normal function, a class method, or a class which implements the __call__() method (AKA callable class, as discussed earlier).

A simple decorator expects the item being decorated as its parameter, and returns a replacement. Typically, the replacement is a new function, but there is no restriction on what is returned. If the decorator itself needs parameters, then the decorator returns a wrapper function that expects the item being decorated, and then returns the replacement.

Table 17. Decorators in the standard library

Decorator	Description
`@abc.abstractmethod`	Indicate abstract method (must be implemented).
`@abc.abstractproperty`	Indicate abstract property (must be implemented). **DEPRECATED**
`@asyncio.coroutine`	Mark generator-based coroutine.
`@atexit.register`	Register function to be executed when interpreter (script) exits.
`@classmethod`	Indicate class method (receives class object, not instance object)
`@contextlib.contextmanager`	Define factory function for **with** statement context managers (no need to create __enter__() and __exit__() methods)
`@functools.lru_cache`	Wrap a function with a memoizing callable
`@functools.singledispatch`	Transform function into a single-dispatch generic function.
`@functools.total_ordering`	Supply all other comparison methods if class defines at least one.
`@functools.wraps`	Invoke update_wrapper() so decorator's replacement function keeps original function's name and other properties.
`@property`	Indicate a class property.
`@staticmethod`	Indicate static method (passed neither instance nor class object).
`@types.coroutine`	Transform generator function into a coroutine function.
`@unittest.mock.patch`	Patch target with a new object. When the function/with statement exits patch is undone.
`@unittest.mock.patch.dict`	Patch dictionary (or dictionary-like object), then restore to original state after test.
`@unittest.mock.patch.multiple`	Perform multiple patches in one call.
`@unittest.mock.patch.object`	Patch object attribute with mock object.
`@unittest.skip()`	Skip test unconditionally
`@unittest.skipIf()`	Skip test if condition is true
`@unittest.skipUnless()`	Skip test unless condition is true
`@unittest.expectedFailure()`	Mark Test as expected failure
`@unittest.removeHandler()`	Remove Control-C handler

Applying decorators

- Use @ symbol
- Applied to *next* item only
- Multiple decorators OK

The **@** sign is used to apply a decorator to a function or class. A decorator only applies to the next definition in the script.

The most important thing to know about the decorators is the following syntax:

```
@spam
def ham():
    pass
```

is exactly the same as

```
ham = spam(ham)
```

and

```
@spam(a, b, c)
def ham():
    pass
```

is exactly the same as

```
ham = spam(a, b, c)(ham)
```

Once you understand this, then creating decorators is just a matter of writing functions or classes and having them return the appropriate thing.

Table 18. Implementing Decorators

Implemented as	**Decorates**	**Takes parameters**	**How to do it**
function	function	N	decorator function returns replacement function
function	function	Y	decorator function accept params and returns function that returns replacement function
function	class	N	decorator function returns replacement class
function	class	Y	decorator function accepts params and returns function that returns replacement class
class	function	N	*instance*.__call__() *is* replacement function
class	function	Y	*instance*.__call__() accepts params and *returns* replacement function
class	class	N	*instance*.__call__() accepts original class, returns replacement class (which is usually same as orginal class)
class	class	Y	*instance*.__call__() accepts params and returns function that returns replacement class

Trivial Decorator

- Decorator can return anything
- Not very useful, usually

A decorator does not have to be elaborate. It can return anything, though typically decorators return the same type of object they are decorating.

In this example, the decorator returns the integer value 42. This is not particularly useful, but illustrates that the decorator always replaces the object being decorated with *something*.

Example

deco_trivial.py

```
def void(thing_being_decorated):
    return 42  # replace function with 42

name = "Guido"
x = void(name)

@void  # decorate hello() function
def hello():
    print("Hello, world")

@void
def howdy():
    print("Howdy, world")

print(hello, type(hello)) # hello is now the integer 42, not a function
print(howdy, type(howdy)) # hello is now the integer 42, not a function
print(x, type(x))
```

deco_trivial.py

```
42 <class 'int'>
42 <class 'int'>
42 <class 'int'>
```

Decorator functions

- Provide a wrapper around a function
- Purposes
  - Add functionality
  - Register
  - ??? (open-ended)
- Optional parameters

A decorator function acts as a wrapper around some object (usually function or class). It allows you to add features to a function without changing the function itself. For instance, the @property, @classmethod, and @staticmethod decorators are used in classes.

A simple decorator function expects only one argument – the function to be modified. It should return a new function, which will replace the original. The replacement function typically calls the original function as well as some new code. More complex decorators expect parameters to the decorator itself. In this case the decorator returns a function that expects the original function, and returns the replacement function.

The new function should be defined with generic arguments (*args, **kwargs) so it can handle any combination of arguments for the original function.

The **wraps** decorator from the functools module in the standard library should be used with the function that returns the replacement function. This makes sure the replacement function keeps the same properties (especially the name) as the original (target) function. Otherwise, the replacement function keeps all of its own attributes.

Example

deco_debug.py

```python
from functools import wraps


def debugger(old_func):  # decorator function -- expects decorated (original) function as
a parameter

    @wraps(old_func)  # @wraps preserves name of original function after decoration
    def new_func(*args, **kwargs):  # replacement function; takes generic parameters
        print("*" * 40)  # new functionality added by decorator
        print("** function", old_func.__name__, "**")  # new functionality added by
decorator

        if args:  # new functionality added by decorator
            print("\targs are ", args)
        if kwargs:  # new functionality added by decorator
            print("\tkwargs are ", kwargs)

        print("*" * 40)  # new functionality added by decorator

        return old_func(*args, **kwargs)  # call the original function

    return new_func  # return the new function object


@debugger  # apply the decorator to a function
def hello(greeting, whom='world'):
    print("{}, {}".format(greeting, whom))


hello('hello', 'world')  # call new function
print()

hello('hi', 'Earth')
print()

hello('greetings')
```

deco_debug.py

```
****************************************
** function hello **
    args are  ('hello', 'world')
****************************************
hello, world

****************************************
** function hello **
    args are  ('hi', 'Earth')
****************************************
hi, Earth

****************************************
** function hello **
    args are  ('greetings',)
****************************************
greetings, world
```

Decorator Classes

- Same purpose as decorator functions
- Two ways to implement
    - No parameters
    - Expects parameters
- Decorator can keep state

A class can also be used to implement a decorator. The advantage of using a class for a decorator is that a class can keep state, so that the replacement function can update information stored at the class level.

Implementation depends on whether the decorator needs parameters.

If the decorator does *not* need parameters, the class must implement two methods: __init__() is passed the original function, and can perform any setup needed. The __call__ method *replaces* the original function. In other word, after the function is decorated, calling the function is the same as calling *CLASS*.__call__.

If the decorator *does* need parameters, __init__() is passed the parameters, and __call__() is passed the original function, and must *return* the replacement function.

A good use for a decorator class is to log how many times a function has been called, or even keep track of the arguments it is called with (see example for this).

Example

deco_debug_class.py

```
class debugger():  # class implementing decorator

    function_calls = []

    def __init__(self, func):  # original function passed into decorator's constructor
        self._func = func

    def __call__(self, *args, **kwargs):  # __call__() is replacement function

        # print("*" * 40)  # add useful features to original function
        # print("function {}()".format(self._func.__name__))  # add useful features to
original function
        # print("\targs are ", args)  # add useful features to original function
        # print("\tkwargs are ", kwargs)  # add useful features to original function
        #
        # print("*" * 40)  # add useful features to original function

        self.function_calls.append(  # add function name and arguments to saved list
            (self._func.__name__, args, kwargs)
        )

        result = self._func(*args, **kwargs)  # call the original function
        return result # return result of calling original function

    @classmethod
    def get_calls(cls): # define method to get saved function call information
        return cls.function_calls

@debugger  # apply debugger to function
def hello(greeting, whom="world"):
    print("{}, {}".format(greeting, whom))

@debugger  # apply debugger to function
def bark(bark_word, *, repeat=2):
    print("{0}! ".format(bark_word) * repeat)

hello('hello', 'world')  # call replacement function
print()

hello('hi', 'Earth')
print()

hello('greetings')
```

```
bark("woof", repeat=3)
bark("yip", repeat=4)
bark("arf")

hello('hey', 'girl')

print('-' * 60)

for i, info in enumerate(debugger.get_calls(), 1):  # display function call info from
class
    print("{:2d}. {:10s} {!s:20s} {!s:20s}".format(i, info[0], info[1], info[2]))
```

deco_debug_class.py

```
hello, world

hi, Earth

greetings, world
woof! woof! woof!
yip! yip! yip! yip!
arf! arf!
hey, girl
------------------------------------------------------------
 1. hello      ('hello', 'world')   {}
 2. hello      ('hi', 'Earth')      {}
 3. hello      ('greetings',)       {}
 4. bark       ('woof',)            {'repeat': 3}
 5. bark       ('yip',)             {'repeat': 4}
 6. bark       ('arf',)             {}
 7. hello      ('hey', 'girl')      {}
```

Decorator parameters

- Decorator functions require two nested functions
- Method __call__() returns replacement function in classes

A decorator can be passed parameters. This requires a little extra work.

For decorators implemented as functions, the decorator itself is passed the parameters; it contains a nested function that is passed the decorated function (the target), and it returns the replacement function.

For decorators implemented as classes, init is passed the parameters, __call__() is passed the decorated function (the target), and __call__ returns the replacement function.

There are many combinations of decorators (8 total, to be exact). This is because decorators can be implemented as either functions or classes, they may take parameters, or not, and they can decorate either functions or classes. For an example of all 8 approaches, see the file **decorama.py** in the EXAMPLES folder.

Example

deco_params.py

```
#

from functools import wraps  # wrapper to preserve properties of original function


def multiply(multiplier): # actual decorator -- receives decorator parameters

    def deco(old_func): # "inner decorator" -- receives function being decorated

        @wraps(old_func)  # retain name, etc. of original function
        def new_func(*args, **kwargs): # replacement function -- this is called instead
of original
            result = old_func(*args, **kwargs) # call original function and get return
value
            return result * multiplier # multiple result of original function by
multiplier

        return new_func # deco() returns new_function

    return deco # multiply returns deco



@multiply(4)
def spam():
    return 5


@multiply(10)
def ham():
    return 8

a = spam()
b = ham()
print(a, b)
```

deco_params.py

```
20 80
```

Creating classes at runtime

- Use the **type()** function
- Provide dictionary of attributes

A class can be created programmatically, without the use of the class statement. The syntax is

```
type("name", (base_class, …), {attributes})
```

The first argument is the name of the class, the second is a tuple of base classes (use object if you are not inheriting from a specific class), and the third is a dictionary of the class's attributes.

NOTE Instead of type, any other *metaclass* can be used.

Example

creating_classes.py

```python
def function_1(self):  # create method (not inside a class -- could be a lambda)
    print("Hello from f1()")


def function_2(self):  # create method (not inside a class -- could be a lambda)
    print("Hello from f2()")


NewClass = type("NewClass", (), {  # create class using type() -- parameters are class
name, base classes, dictionary of attributes
    'hello1': function_1,
    'hello2': function_2,
    'color': 'red',
    'state': 'Ohio',
})

n1 = NewClass()  # create instance of new class

n1.hello1()  # call instance method
n1.hello2()
print(n1.color)  # access class data
print()

SubClass = type("SubClass", (NewClass,), {'fruit': 'banana'})  # create subclass of first
class
s1 = SubClass()  # create instance of subclass
s1.hello1()  # call method on subclass
print(s1.color)  # access class data
print(s1.fruit)
```

creating_classes.py

```
Hello from f1()
Hello from f2()
red

Hello from f1()
red
banana
```

Monkey Patching

- Modify existing class or object
- Useful for enabling/disabling behavior
- Can cause problems

"Monkey patching" refers to technique of changing the behavior of an object by adding, replacing, or deleting attributes from outside the object’s class definition.

It can be used for:

- Replacing methods, attributes, or functions
- Modifying a third-party object for which you do not have access
- Adding behavior to objects in memory

If you are not careful when creating monkey patches, some hard-to-debug problems can arise

- If the object being patched changes after a software upgrade, the monkey patch can fail in unexpected ways.
- Conflicts may occur if two different modules monkey-patch the same object.
- Users of a monkey-patched object may not realize which behavior is original and which comes from the monkey patch.

Monkey patching defeats object encapsulation, and so should be used sparingly.

Decorators are a convenient way to monkey-patch a class. The decorator can just add a method to the decorated class.

Example

meta_monkey.py

```
class Spam():  # create normal class

    def __init__(self, name):
        self._name = name

    def eggs(self):  # add normal method
        print("Good morning, {}. Here are your delicious fried eggs.".format(self._name,
))


s = Spam('Mrs. Higgenbotham')  # create instance of class
s.eggs()  # call method


def scrambled(self):  # define new method outside of class
    print("Hello, {}. Enjoy your scrambled eggs".format(self._name, ))


setattr(Spam, "eggs", scrambled)  # monkey patch the class with the new method

s.eggs()  # call the monkey-patched method from the instance
```

meta_monkey.py

```
Good morning, Mrs. Higgenbotham. Here are your delicious fried eggs.
Hello, Mrs. Higgenbotham. Enjoy your scrambled eggs
```

Do you need a Metaclass?

- Deep magic
- Used in frameworks such as Django
- YAGNI (You Ain't Gonna Need It)

Before we cover the details of metaclasses, a disclaimer: you will probably never need to use a metaclass. When you think you might need a metaclass, consider using inheritance or a class decorator. However, metaclasses may be a more elegant approach to certain kinds of tasks, such as registering classes when they are defined.

There are two use cases where metaclasses are always an appropriate solution, because they must be done before the class is created:

- Modifying the class name
- Modifying the the list of base classes.

Several popular frameworks use metaclasses, Django in particular. In Django they are used for models, forms, form fields, form widgets, and admin media.

Remember that metaclasses can be a more elegant way to accomplish things that can also be done with inheritance, composition, decorators, and other techniques that are less "magic".

About metaclasses

- Metaclass:Class::Class:Object

Just as a class is used to create an instance, a *metaclass* is used to create a class.

The primary reason for a metaclass is to provide extra functionality at *class creation* time, not *instance creation* time. Just as a class can share state and actions across many instances, a metaclass can share (or provide) data and state across many *classes*.

The metaclass might modify the list of base classes, or register the class for later retrieval.

The builtin metaclass that Python provides is **type**.

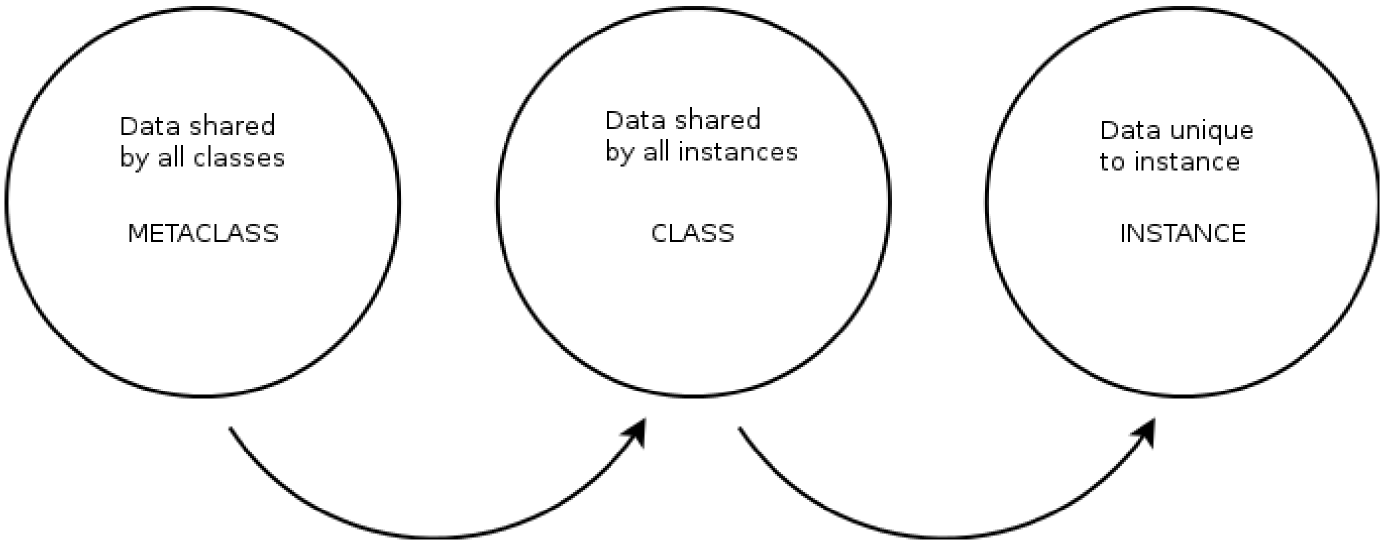
As we saw earlier ,you can create a class from a metaclass by passing in the new class's name, a tuple of base classes (which can be empty), and a dictionary of class attributes (which also can be empty).

```
class Spam(Ham):
    id = 1
```

is exactly equivalent to

```
Spam = type('Spam', (Ham,), {"id": 1})
```

Replacing "type" with the name of any other metaclass works the same.



Mechanics of a metaclass

- Like normal class
- *Should* implement __new__
- *Can* implement
    - __init__
    - __prepare__
    - __call__

To create a metaclass, define a normal class. Most metaclasses implement the **__new__** method. This method is called with the type, name, base classes, and attribute dictionary (if any) of the new class. It should return a new class, typically using **super().__new__()**, which is very similar to how normal classes create instances. This is one place you can modify the class being created. You can add or change attributes, methods, or properties.

For instance, the Django framework uses metaclasses for Models. When you create an instance of a Model, the metaclass code automatically creates methods for the fields in the model. This is called "declarative programming", and is also used in SqlAlchemy's declarative model, in a way pretty similar to Django.

When you execute the following code:

```
class SomeClass(metaclass=SomeMeta):
    pass
```

META(name, bases, attrs) is executed, where META is the metaclass (normally **type()**). Then,

1. The __prepare__ method of the metaclass is called
2. The __new__ method of the metaclass is called
3. The __init__ method of the metaclass is called.

Next, after the following code runs:

```
obj = SomeClass()
```

SomeMeta.call() is called. It returns whatever SomeMeta.__new__() returned.

__prepare__() __new__() __init__() __call__()

Example

metaclass_generic.py

```
class Meta(type):

    def __prepare__(class_name, bases):
        """
        "Prepare" the new class. Here you can update the base classes.

        :param name: Name of new class as a string
        :param bases: Tuple of base classes
        :return: Dictionary that initializes the namespace for the new class (must be a
dict)
        """
        print("in metaclass (class={}) __prepare__()".format(class_name), end=' ==> ')
        print("params: name={}, bases={}".format(class_name, bases))
        return {'animal': 'wombat', 'id': 100}

    def __new__(metatype, name, bases, attrs):
        """
        Create the new class. Called after __prepare__(). Note this is only called when
classes

        :param metatype: The metaclass itself
        :param name: The name of the class being created
        :param bases: bases of class being created (may be empty)
        :param attrs: Initial attributes of the class being created
        :return:
        """
        print("in metaclass (class={}) __new__()".format(name), end=' ==> ')
        print("params: type={} name={} bases={} attrs={}".format(metatype, name, bases,
attrs))
        return super().__new__(metatype, name, bases, attrs)

    def __init__(cls, *args):
        """

        :param cls: The class being created (compare with 'self' in normal class)
        :param args: Any arguments to the class
        """
        print("in metaclass (class={}) __init__()".format(cls.__name__), end=' ==> ')
        print("params: cls={}, args={}".format(cls, args))

        super().__init__(cls)

    def __call__(self, *args, **kwargs):
        """
        Function called when the metaclass is called, as in NewClass = Meta(...)

        :param args:
        :param args:
        :param kwargs:
        :return:
        """
        print("in metaclass (class={})__call__()".format(self.__name__))


class MyBase():
    pass

print('=' * 60)

class A(MyBase, metaclass=Meta):
    id = 5

    def __init__(self):
        print("In class A __init__()")

print('=' * 60)

class B(MyBase, metaclass=Meta):
    animal = 'wombat'

    def __init__(self):
        print("In class B __init__()")


print('-' * 60)
m1 = A()
print('-' * 60)
m2 = B()
print('-' * 60)
m3 = A()
print('-' * 60)
m4 = B()
print('-' * 60)
print("animal: {} id: {}".format(A.animal, B.id))
```

metaclass_generic.py

```
============================================================
in metaclass (class=A) __prepare__() ==> params: name=A, bases=(<class
'__main__.MyBase'>,)
in metaclass (class=A) __new__() ==> params: type=<class '__main__.Meta'> name=A
bases=(<class '__main__.MyBase'>,) attrs={'animal': 'wombat', 'id': 5, '__module__':
'__main__', '__qualname__': 'A', '__init__': <function A.__init__ at 0x7ff66029e670>}
in metaclass (class=A) __init__() ==> params: cls=<class '__main__.A'>, args=('A',
(<class '__main__.MyBase'>,), {'animal': 'wombat', 'id': 5, '__module__': '__main__',
'__qualname__': 'A', '__init__': <function A.__init__ at 0x7ff66029e670>})
============================================================
in metaclass (class=B) __prepare__() ==> params: name=B, bases=(<class
'__main__.MyBase'>,)
in metaclass (class=B) __new__() ==> params: type=<class '__main__.Meta'> name=B
bases=(<class '__main__.MyBase'>,) attrs={'animal': 'wombat', 'id': 100, '__module__':
'__main__', '__qualname__': 'B', '__init__': <function B.__init__ at 0x7ff66029e700>}
in metaclass (class=B) __init__() ==> params: cls=<class '__main__.B'>, args=('B',
(<class '__main__.MyBase'>,), {'animal': 'wombat', 'id': 100, '__module__': '__main__',
'__qualname__': 'B', '__init__': <function B.__init__ at 0x7ff66029e700>})
------------------------------------------------------------
in metaclass (class=A)__call__()
------------------------------------------------------------
in metaclass (class=B)__call__()
------------------------------------------------------------
in metaclass (class=A)__call__()
------------------------------------------------------------
in metaclass (class=B)__call__()
------------------------------------------------------------
animal: wombat id: 100
```

Singleton with a metaclass

- Classic example
- Simple to implement
- Works with inheritance

One of the classic use cases for a metaclass in Python is to create a *singleton* class. A singleton is a class that only has one actual instance, no matter how many times it is instantiated. Singletons are used for loggers, config data, and database connections, for instance.

To create a single, implement a metaclass by defining a class that inherits from **type**. The class should have a class-level dictionary to store each class's instance. When a new instance of a class is created, check to see if that class already has an instance. If it does not, call __call__ to create the new instance, and add the instance to the dictionary.

In either case, then return the instance where the key is the class object.

Example

metaclass_singleton.py

```
class Singleton(type): # use type as base class of a metaclas
    _instances = {}  # dictionary to keep track of instances

    def __new__(typ, *junk):
        # print("__new__()")
        return super().__new__(typ, *junk)

    def __call__(cls, *args, **kwargs):  # __call__ is passed the new class plus its
parameters
        # print("__call__()")
        if cls not in cls._instances:    # check to see if the new class has already been
instantiated
            cls._instances[cls] = super().__call__(*args, **kwargs)  # if not, create the
(single) class instance and add to dictionary

        return cls._instances[cls]   # return the (single) class instance


class ThingA(metaclass=Singleton):   # Define two different classes which use Singleton
    def __init__(self, value):
        self.value = value


class ThingB(metaclass=Singleton):   # Define two different classes which use Singleton
    def __init__(self, value):
        self.value = value


ta1 = ThingA(1)  # Create instances of ThingA and ThingB
ta2 = ThingA(2)
ta3 = ThingA(3)

tb1 = ThingB(4)
tb2 = ThingB(5)
tb3 = ThingB(6)

for thing in ta1, ta2, ta3, tb1, tb2, tb3:
    print(type(thing).__name__, id(thing), thing.value)  # Print the type, name, and ID
of each thing -- only one instance is ever created for each class
```

metaclass_singleton.py

```
ThingA 140653199780640 1
ThingA 140653199780640 1
ThingA 140653199780640 1
ThingB 140653199780400 4
ThingB 140653199780400 4
ThingB 140653199780400 4
```

Chapter 13 Exercises

Exercise 13-1 (pres_attr.py)

Instantiate the President class. Get the first name, last name, and party attributes using getattr().

Exercise 13-2 (pres_monkey.py, pres_monkey_amb.py)

Monkey-patch the President class to add a method get_full_name which returns a single string consisting of the first name and the last name, separated by a space.

TIP Instead of a method, make full_name a property.

Exercise 13-3 (sillystring.py)

Without using the **class** statement, create a class named SillyString, which is initialized with any string. Include an instance method called every_other which returns every other character of the string.

Instantiate your string and print the result of calling the **every_other()** method. Your test code should look like this:

```
ss = SillyString('this is a test')
print(ss.every_other())
```

It should output

```
ti sats
```

Exercise 13-4 (doubledeco.py)

Write a decorator to double the return value of any function. If a function returns 5, after decoration it should return 10. If it returns "spam", after decoration it should return "spamspam", etc.

Exercise 13-5 (word_actions.py)

Write a decorator, implemented as a class, to register functions that will process a list of words. The decorated functions will take one parameter — a string — and return the modified string.

The decorator itself takes two parameters — minimum length and maximum length. The class will store the min/max lengths as the key, and the functions as values, as class data.

The class will also provide a method named **process_words**, which will open **DATA/words.txt** and read it line by line. Each line contains a word.

For every registered function, if the length of the current word is within the min/max lengths, call all the functions whose key is that min/max pair.

In other words, if the registry key is (5, 8), and the value is [func1, func2], when the current word is within range, call func1(*w*) and func2(*w*), where *w* is the current word.

Example of class usage:

```
word_select = WordSelect()  # create callable instance

@word_select(16, 18) # register function for length 16-18, inclusive
def make_upper(s):
    return s.upper()

word_select.process_words()  # loop over words, call functions if selected
```

Suggested functions to decorate:

- make the word upper-case
- put stars before or around the word
- reverse the word

Remember all the decorated functions take one argument, which is one of the strings in the word list, and return the modified word.

Chapter 14: Introduction to NumPy

Objectives

- See the "big picture" of NumPy
- Create and manipulate arrays
- Learn different ways to initialize arrays
- Understand the NumPy data types available
- Work with shapes, dimensions, and ranks
- Broadcast changes across multiple array dimensions
- Extract multidimensional slices
- Perform matrix operations

Python’s scientific stack

- NumPy, SciPy, MatPlotLib (and many others)
- Python extensions, written in C/Fortran
- Support for math, numerical, and scientific operations

NumPy is part of what is sometimes called Python’s "scientific stack". Along with SciPy, Matplotlib, and other libraries, it provides a broad range of support for scientific and engineering tasks.

SciPy is a large group of mathematical algorithms, along with some convenience functions, for doing scientific and engineering calculations, including data science. SciPy routines accept and return NumPy arrays.

pandas ties some of the libraries together, and is frequently used interactively via **iPython** in a **Jupyter** notebook. Of course you can also create scripts using any of the scientific libraries.

NOTE There is not an integrated *application* for all of the Python scientific libraries.

NumPy overview

- Install numpy module from numpy.scipy.org (included with Anaconda)
- Basic object is the array
- Up to 100x faster than normal Python math operations
- Functional-based (fewer loops)
- Dozens of utility functions

The basic object that NumPy provides is the array. Arrays can have as many dimensions as needed. Working with NumPy arrays can be 100 times faster than working with normal Python lists.

Operations are applied to arrays in a functional manner – instead of the programmer explicitly looping through elements of the array, the programmer specifies an expression or function to be applied, and the array object does all of the iteration internally.

There are many utility functions for accessing arrays, for creating arrays with specified values, and for performing standard numerical operations on arrays.

To get started, import the **numpy** module. It is conventional to import numpy as **np**. The examples in this chapter will follow that convention.

NumPy and the rest of the Python scientific stack is included with the Anaconda, Canopy, Python(x,y), and WinPython bundles. If you are not using one of these, install NumPy with

```
pip install numpy
```

NOTE all top-level NumPy routines are also available directly through the scipy package.

Creating Arrays

- Create with
  - array() function initialized with nested sequences
  - Other utilities (arange(), zeros(), ones(), empty()
- All elements are same type (default float)
- Useful properties: ndim, shape, size, dtype
- Can have any number of axes (dimensions)
- Each axis has a length

An array is the most basic object in NumPy. It is a table of numbers, indexed by positive integers. All of the elements of an array are of the same type.

An array can have any number of dimensions; these are referred to as axes. The number of axes is called the rank.

Arrays are rectangular, not ragged.

One way to create an array is with the `array()` function, which can be initialized from existing arrays.

The `zeros()` function expects a *shape* (tuple of axis lengths), and creates the corresponding array, with all values set to zero. The `ones()` function is the same, but initializes with ones.

The `full()` function expects a shape and a value. It creates the array, putting the specified value in every element.

The `empty()` function creates an array of specified shape initialized with random floats.

However, the most common way to crate an array is by loading data from a text or binary file.

When you print an array, NumPy displays it with the following layout:

- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

NOTE the `ndarray()` object is initialized with the *shape*, not the *data*.

Example

np_create_arrays.py

```
import sys
import numpy as np
data = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [20, 30, 40]]

a = np.array(data)  # create array from nested sequences
print(a, '\n')

print("a.ndim (# dimensions):", a.ndim)  # get number of dimensions
print("a.shape (lengths of axes/dimensions):", a.shape)  # get shape
print("a.size (number of elements in array):", a.size)
print("a.itemsize (size of one item):", a.itemsize)
print("a.nbytes (number of bytes used):", a.nbytes)
print("sys.getsizeof(data):", sys.getsizeof(data))
print()

a_zeros = np.zeros((3, 5), dtype=np.uint32)  # create array of specified shape and
datatype, initialized to zeroes
print(a_zeros)
print()

a_ones = np.ones((2, 3, 4, 5))  # create array of specified shape, initialized to ones
print(a_ones)
print()

# with uninitialized values
a_empty = np.empty((3, 8))  # create uninitialized array of specified shape
print(a_empty)

print(a.dtype)  # defaults to float64

nan_array = np.full((5, 10), np.NaN)  # create array of NaN values
print(nan_array)
```

np_create_arrays.py

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [20 30 40]]

a.ndim (# dimensions): 2
a.shape (lengths of axes/dimensions): (4, 3)
a.size (number of elements in array): 12
a.itemsize (size of one item): 8
a.nbytes (number of bytes used): 96
sys.getsizeof(data): 88

[[0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]

[[[[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]

  [[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]

  [[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]]


 [[[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]

  [[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]

  [[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]]]

[[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  4.78344603e-01
   8.50390406e-01  1.20419856e+00 -5.73695579e-01 -2.62086512e-01]
 [ 2.24225511e-01 -2.54675889e-01  3.27915049e-01  4.25118474e-01
   2.00133715e-01  4.70330436e-01  4.70601113e-01  1.41111658e+01]
 [ 5.47722558e+00  2.23606798e+00  1.11022302e-16 -6.66133815e-16
   1.63437886e+00  5.59500345e-01  1.25559236e-01  0.00000000e+00]]
int64
[[nan nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan nan]]
```

Creating ranges

- Similar to builtin `range()`
- Returns a one-dimensional NumPy array
- Can use floating point values
- Can be reshaped

The `arange()` function takes a size, and returns a one-dimensional NumPy array. This array can then be reshaped as needed. The start, stop, and step parameters are similar to those of `range()`, or Python slices in general. Unlike the builtin Python `range()`, start, stop, and step can be floats.

The `linspace()` function creates a specified number of equally-spaced values. As with `numpy.arange()`, start and stop may be floats.

The resulting arrays can be reshaped into multidimensional arrays.

Example

np_create_ranges.py

```
import numpy as np

r1 = np.arange(50)  # create range of ints from 0 to 49
print(r1)
print("size is", r1.size)  # size is 50
print()

r2 = np.arange(5, 101, 5)  # create range of ints from 5 to 100 counting by 5
print(r2)
print("size is", r2.size)
print()

r3 = np.arange(1.0, 5.0, .3333333)  # start, stop, and step may be floats
print(r3)
print("size is", r3.size)
print()

r4 = np.linspace(1.0, 2.0, 10)  # 10 equal steps between 1.0 and 2.0
print(r4)
print("size is", r4.size)
print()
```

np_create_ranges.py

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49]
size is 50

[  5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90
  95 100]
size is 20

[1.        1.3333333 1.6666666 1.9999999 2.3333332 2.6666665 2.9999998
 3.3333331 3.6666664 3.9999997 4.333333  4.6666663 4.9999996]
size is 13

[1.         1.11111111 1.22222222 1.33333333 1.44444444 1.55555556
 1.66666667 1.77777778 1.88888889 2.        ]
size is 10
```

Working with arrays

- Use normal math operators (+, -, /, and *)
- Use NumPy's builtin functions
- By default, apply to every element
- Can apply to single axis
- Operations on between two arrays applies operator to pairs of element

The array object is smart about applying functions and operators. A function applied to an array is applied to every element of the array. An operator applied to two arrays is applied to corresponding elements of the two arrays.

In-place operators (**+=**, ***=**, etc) efficiently modify the array itself, rather than returning a new array.

Example

np_basic_array_ops.py

```
import numpy as np

a = np.array(
    [
        [5, 10, 15],
        [2, 4, 6],
        [3, 6, 9, ],
    ]
)  # create 2D array

b = np.array(
    [
        [10, 85, 92],
        [77, 16, 14],
        [19, 52, 23],
    ]
)  # create another 2D array
print("a")
print(a)
print()
print("b")
print(b)
print()

print("a * 10")
print(a * 10)  # multiply every element by 10 (not in place)
print()

print("a + b")
print(a + b)  # add every element of a to the corresponding element of b
print()

print("b + 3")
print(b + 3)  # add 3 to every element of b
print()

print("a.sum(): {}".format(a.sum()))
print("a.std(): {}".format(a.std()))
print("a.mean(): {}".format(a.mean()))
print("a.cumsum(): {}".format(a.cumsum()))
print("a.cumprod(): {}".format(a.cumprod()))


def c2f(cel):  # user-defined function
    return (9/5 * cel) + 32

f_temps = c2f(a)  # apply function to elements of a
print("f_temps:\n", f_temps)

print()
a += 1000  # add 1000 to every element of a (in place)
print("a after 'a += 1000'")
print(a)
```

np_basic_array_ops.py

```

a
[[ 5 10 15]
 [ 2  4  6]
 [ 3  6  9]]

b
[[10 85 92]
 [77 16 14]
 [19 52 23]]

a * 10
[[ 50 100 150]
 [ 20  40  60]
 [ 30  60  90]]
```

```
a + b
[[ 15  95 107]
 [ 79  20  20]
 [ 22  58  32]]

b + 3
[[13 88 95]
 [80 19 17]
 [22 55 26]]

a.sum(): 60
a.std(): 3.8297084310253524
a.mean(): 6.666666666666667
a.cumsum(): [ 5 15 30 32 36 42 45 51 60]
a.cumprod(): [      5      50     750    1500    6000   36000  108000  648000 5832000]
f_temps:
 [[41.  50.  59. ]
 [35.6 39.2 42.8]
 [37.4 42.8 48.2]]

a after 'a += 1000'
[[1005 1010 1015]
 [1002 1004 1006]
 [1003 1006 1009]]
```

Shapes

- Number of elements on each axis
- array.shape has shape tuple
- Assign to array.shape to change
- Convert to one dimension
    - array.ravel()
    - array.flatten()
- array.transpose() to flip the shape

Every array has a shape, which is the number of elements on each axis. For instance, an array might have the shape (3,5), which means that there are 3 rows and 5 columns.

The shape is stored as a tuple, in the shape attribute of an array. To change the shape of an array, assign to the shape attribute.

The `ravel()` and `flatten()` methods will flatten any array into a single dimension. `ravel()` returns a "view" of the original array, while `flatten()` returns a new array. If you modify the result of ravel(), it will modify the original data.

The `transpose()` method will flip shape (x,y) to shape (y,x). It is equivalent to `array.shape = list(reversed(array.shape))`.

Example

np_shapes.py

```python
import numpy as np

a1 = np.arange(15)  # create 1D array
print("a1 shape", a1.shape)  # get shape
print()

print(a1)
print()

a1.shape = 3, 5  # reshape to 3x5
print(a1)
print()

a1.shape = 5, 3  # reshape to 5x3
print(a1)
print()

print(a1.flatten())  # print array as 1D
print()

print(a1.transpose())  # print transposed array
print("------------------")

a2 = np.arange(40)  # create 1D array
a2.shape = 2, 5, 4  # reshape to 2x5x4

print(a2)
print()
```

np_shapes.py

```
a1 shape (15,)

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]

[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]

[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]

[[ 0  3  6  9 12]
 [ 1  4  7 10 13]
 [ 2  5  8 11 14]]
------------------
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]
  [12 13 14 15]
  [16 17 18 19]]

 [[20 21 22 23]
  [24 25 26 27]
  [28 29 30 31]
  [32 33 34 35]
  [36 37 38 39]]]
```

Slicing and indexing

- Simple indexing similar to lists
- start, stop, step
- start is INclusive, stop is Exclusive
- : used for range for one axis
- … means "as many : as needed"

NumPy arrays can be indexed and sliced like regular Python lists, but with some convenient extensions. Instead of `[x][y]`, NumPy arrays can be indexed with `[x,y]`. Within an axis, ranges can be specified with slice notation (start:stop:step) as usual.

For arrays with more than 2 dimensions, `…` can be used to mean "and all the other dimensions".

Example

np_indexing.py

```python
import numpy as np

a = np.array(
    [[70, 31, 21, 76, 19, 5, 54, 66],
     [23, 29, 71, 12, 27, 74, 65, 73],
     [11, 84, 7, 10, 31, 50, 11, 98],
     [25, 13, 43, 1, 31, 52, 41, 90],
     [75, 37, 11, 62, 35, 76, 38, 4]]
)  # sample data

print(a)
print()

print('a[0] =>', a[0])  # first row
print('a[0][0] =>', a[0][0])  # first element of first row
print('a[0,0] =>', a[0, 0])  # same, but numpy style
print('a[0,:3] =>', a[0, :3])  # first 3 elements of first row
print('a[0,::2] =>', a[0, ::2])  # every second element of first row
print()
print('a[::2] =>', a[::2])  # every second row
print()
print('a[:3, -2:] =>', a[:3, -2:])  # every third element of every second row
```

np_indexing.py

```
[[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]
 [11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]
 [75 37 11 62 35 76 38  4]]

a[0] => [70 31 21 76 19  5 54 66]
a[0][0] => 70
a[0,0] => 70
a[0,:3] => [70 31 21]
a[0,::2] => [70 21 19 54]

a[::2] => [[70 31 21 76 19  5 54 66]
 [11 84  7 10 31 50 11 98]
 [75 37 11 62 35 76 38  4]]

a[:3, -2:] => [[54 66]
 [65 73]
 [11 98]]
```

Indexing with Booleans

- Apply relational expression to array
- Result is array of Booleans
- Booleans can be used to index original array

If a relational expression (>, <, >=, ⇐) is applied to an array, the result is a new array containing Booleans reflecting whether the expression was true for each element. That is, for each element of the original array, the resulting array is set to True if the expression is true for that element, and False otherwise.

The resulting Boolean array can then be used as an index, to modify just the elements for which the expression was true.

Example

np_bool_indexing.py

```
import numpy as np

a = np.array(
    [[70, 31, 21, 76, 19, 5, 54, 66],
     [23, 29, 71, 12, 27, 74, 65, 73],
     [11, 84, 7, 10, 31, 50, 11, 98],
     [25, 13, 43, 1, 31, 52, 41, 90],
     [75, 37, 11, 62, 35, 76, 38, 4]]
)  # sample data

print('a =>', a, '\n')

i = a > 50  # create Boolean mask
print('i (a > 50) =>', i, '\n')

print('a[i] =>', a[i], '\n')  # print elements of a that are > 50 using mask

print('a[a > 50] =>', a[a > 50], '\n')  # same, but without creating a separate mask

print('a[i].min(), a[i].max() =>', a[i].min(), a[i].max(), '\n')  # min and max values of
result set with values less than 50

a[i] = 0  # set elements with value > 50 to 0
print('a =>', a, '\n')

print("a[a < 15] += 10")
a[a < 15] += 10  # add 10 to elements < 15
print(a, '\n')
```

np_bool_indexing.py

```
a => [[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]
 [11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]
 [75 37 11 62 35 76 38  4]]

i (a > 50) => [[ True False False  True False False  True  True]
 [False False  True False False  True  True  True]
 [False  True False False False False False  True]
 [False False False False False  True False  True]
 [ True False False  True False  True False False]]

a[i] => [70 76 54 66 71 74 65 73 84 98 52 90 75 62 76]

a[a > 50] => [70 76 54 66 71 74 65 73 84 98 52 90 75 62 76]

a[i].min(), a[i].max() => 52 98

a => [[ 0 31 21  0 19  5  0  0]
 [23 29  0 12 27  0  0  0]
 [11  0  7 10 31 50 11  0]
 [25 13 43  1 31  0 41  0]
 [ 0 37 11  0 35  0 38  4]]

a[a < 15] += 10
[[10 31 21 10 19 15 10 10]
 [23 29 10 22 27 10 10 10]
 [21 10 17 20 31 50 21 10]
 [25 23 43 11 31 10 41 10]
 [10 37 21 10 35 10 38 14]]
```

Selecting rows based on conditions

- Index with boolean expressions
- Use **&**, not **and**

To select rows from an array, based on conditions, you can index the array with two or more Boolean expressions.

Since the Boolean expressions return arrays of True/False values, use the **&** bitwise AND operator (or **|** for OR).

Any number of conditions can be applied this way.

```
new_array = old_array[bool_expr1 & bool_expr2 ...]
```

Example

np_select_rows.py

```
import numpy as np

sample_data = np.loadtxt(   # Read some data into 2d array
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
)

print("first 5 rows of sample_data:")
print(sample_data[:5, :], '\n')

selected = sample_data[  # Index into the existing data
    (sample_data[:, 0] < 10) &  # Combine two Boolean expressions with &
    (sample_data[:, -1] > 35)
]

print("selected")
print(selected)
```

np_select_rows.py

```
first 5 rows of sample_data:
[[63. 51. 59. 61. 50.  4.]
 [40. 66.  9. 64. 63. 17.]
 [18. 23.  2. 61.  1.  9.]
 [29.  8. 40. 59. 10. 26.]
 [54.  9. 68.  4. 16. 21.]]

selected
[[ 8. 49.  2. 40. 50. 36.]
 [ 4. 49. 39. 50. 23. 39.]
 [ 6.  7. 40. 56. 31. 38.]
 [ 6.  1. 44. 55. 49. 36.]
 [ 5. 22. 45. 49. 10. 37.]]
```

Stacking

- Combining 2 arrays vertically or horizontally
- use `vstack()` or `hstack()`
- Arrays must have compatible shapes

You can combine two or more arrays vertically or horizontally with the `vstack()` or `hstack()` functions. These functions are also handy for adding rows or columns with the results of operations.

Example

np_stacking.py

```
import numpy as np

a = np.array(
    [[70, 31, 21, 76, 19, 5, 54, 66],
     [23, 29, 71, 12, 27, 74, 65, 73]]
)  # sample array a

b = np.array(
    [[11, 84, 7, 10, 31, 50, 11, 98],
     [25, 13, 43, 1, 31, 52, 41, 90]]
)  # sample array b

print('a =>\n', a)
print()
print('b =>\n', b)
print()
print('vstack((a,b)) =>\n', np.vstack((a, b)))  # stack arrays vertically (like pancakes)
print()

print('vstack((a,a[0] + a[1])) =>\n', np.vstack((a, a[0] + a[1])))  # add a row with sums
of first two rows
print()

print('hstack((a,b)) =>\n', np.hstack((a, b)))  # stack arrays horizontally (like books
on a shelf)
```

np_stacking.py

```
a =>
 [[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]]

b =>
 [[11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]]

vstack((a,b)) =>
 [[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]
 [11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]]

vstack((a,a[0] + a[1])) =>
 [[ 70  31  21  76  19   5  54  66]
 [ 23  29  71  12  27  74  65  73]
 [ 93  60  92  88  46  79 119 139]]

hstack((a,b)) =>
 [[70 31 21 76 19  5 54 66 11 84  7 10 31 50 11 98]
 [23 29 71 12 27 74 65 73 25 13 43  1 31 52 41 90]]
```

ufuncs and builtin operators

- Builtin functions for efficiency
- Map over array
- No **for** loops
- Use **vectorize()** for custom ufuncs

In normal Python, you are used to iterating over arrays, especially nested arrays, with a **for** loop. However, for large amounts of data, this is slow. The reason is that the interpreter must do type-checking and lookups for each item being looped over.

NumPy provides *vectorized* operations which are implemented by *ufuncs* — universal functions. ufuncs are implemented in C and work directly on NumPy arrays. When you use a normal math operator (**+ - * /**, etc) on a NumPy array, it calls the underlying ufunc. For instance, `array1 + array2` calls `np.add(array1, array2)`.

There are over 60 ufuncs built into NumPy. These normally return a NumPy array with the results of the operation. Some have options for putting the output into a different object.

The official docs for ufuncs are here:

https://numpy.org/doc/stable/reference/ufuncs.html#available-ufuncs

You can scroll down to the list of available ufuncs.

Table 19. List of NumPy universal functions (ufunc)

Math operations	
`add(x1, x2, /[, out, where, casting, order, …])`	Add arguments element-wise.
`subtract(x1, x2, /[, out, where, casting, …])`	Subtract arguments, element-wise.
`multiply(x1, x2, /[, out, where, casting, …])`	Multiply arguments element-wise.
`divide(x1, x2, /[, out, where, casting, …])`	Returns a true division of the inputs, element-wise.
`logaddexp(x1, x2, /[, out, where, casting, …])`	Logarithm of the sum of exponentiations of the inputs.
`logaddexp2(x1, x2, /[, out, where, casting, …])`	Logarithm of the sum of exponentiations of the inputs in base-2.
`true_divide(x1, x2, /[, out, where, …])`	Returns a true division of the inputs, element-wise.
`floor_divide(x1, x2, /[, out, where, …])`	Return the largest integer smaller or equal to the division of the inputs.
`negative(x, /[, out, where, casting, order, …])`	Numerical negative, element-wise.
`positive(x, /[, out, where, casting, order, …])`	Numerical positive, element-wise.
`power(x1, x2, /[, out, where, casting, …])`	First array elements raised to powers from second array, element-wise.
`remainder(x1, x2, /[, out, where, casting, …])`	Return element-wise remainder of division.
`mod(x1, x2, /[, out, where, casting, order, …])`	Return element-wise remainder of division.
`fmod(x1, x2, /[, out, where, casting, …])`	Return the element-wise remainder of division.
`divmod(x1, x2[, out1, out2], / [[, out, …])`	Return element-wise quotient and remainder simultaneously.
`absolute(x, /[, out, where, casting, order, …])`	Calculate the absolute value element-wise.
`fabs(x, /[, out, where, casting, order, …])`	Compute the absolute values element-wise.
`rint(x, /[, out, where, casting, order, …])`	Round elements of the array to the nearest integer.
`sign(x, /[, out, where, casting, order, …])`	Returns an element-wise indication of the sign of a number.
`heaviside(x1, x2, /[, out, where, casting, …])`	Compute the Heaviside step function.

`conj(x, /[, out, where, casting, order, …])`	Return the complex conjugate, element-wise.
`conjugate(x, /[, out, where, casting, …])`	Return the complex conjugate, element-wise.
`exp(x, /[, out, where, casting, order, …])`	Calculate the exponential of all elements in the input array.
`exp2(x, /[, out, where, casting, order, …])`	Calculate 2**p for all p in the input array.
`log(x, /[, out, where, casting, order, …])`	Natural logarithm, element-wise.
`log2(x, /[, out, where, casting, order, …])`	Base-2 logarithm of x.
`log10(x, /[, out, where, casting, order, …])`	Return the base 10 logarithm of the input array, element-wise.
`expm1(x, /[, out, where, casting, order, …])`	Calculate exp(x) - 1 for all elements in the array.
`log1p(x, /[, out, where, casting, order, …])`	Return the natural logarithm of one plus the input array, element-wise.
`sqrt(x, /[, out, where, casting, order, …])`	Return the non-negative square-root of an array, element-wise.
`square(x, /[, out, where, casting, order, …])`	Return the element-wise square of the input.
`cbrt(x, /[, out, where, casting, order, …])`	Return the cube-root of an array, element-wise.
`reciprocal(x, /[, out, where, casting, …])`	Return the reciprocal of the argument, element-wise.
`gcd(x1, x2, /[, out, where, casting, order, …])`	Returns the greatest common divisor of |x1| and |x2|
`lcm(x1, x2, /[, out, where, casting, order, …])`	Returns the lowest common multiple of |x1| and |x2|
Trigonometric functions These all use radians when an angle is called for. The ratio of degrees to radians is 180°/**pi**.	
`sin(x, /[, out, where, casting, order, …])`	Trigonometric sine, element-wise.
`cos(x, /[, out, where, casting, order, …])`	Cosine element-wise.
`tan(x, /[, out, where, casting, order, …])`	Compute tangent element-wise.
`arcsin(x, /[, out, where, casting, order, …])`	Inverse sine, element-wise.
`arccos(x, /[, out, where, casting, order, …])`	Trigonometric inverse cosine, element-wise.
`arctan(x, /[, out, where, casting, order, …])`	Trigonometric inverse tangent, element-wise.

`arctan2(x1, x2, /[, out, where, casting, …])`	Element-wise arc tangent of x1/x2 choosing the quadrant correctly.
`hypot(x1, x2, /[, out, where, casting, …])`	Given the "legs" of a right triangle, return its hypotenuse.
`sinh(x, /[, out, where, casting, order, …])`	Hyperbolic sine, element-wise.
`cosh(x, /[, out, where, casting, order, …])`	Hyperbolic cosine, element-wise.
`tanh(x, /[, out, where, casting, order, …])`	Compute hyperbolic tangent element-wise.
`arcsinh(x, /[, out, where, casting, order, …])`	Inverse hyperbolic sine element-wise.
`arccosh(x, /[, out, where, casting, order, …])`	Inverse hyperbolic cosine, element-wise.
`arctanh(x, /[, out, where, casting, order, …])`	Inverse hyperbolic tangent element-wise.
`deg2rad(x, /[, out, where, casting, order, …])`	Convert angles from degrees to radians.
`rad2deg(x, /[, out, where, casting, order, …])`	Convert angles from radians to degrees.
Bit-twiddling functions These function all require integer arguments and they manipulate the bit-pattern of those arguments.	
`bitwise_and(x1, x2, /[, out, where, …])`	Compute the bit-wise AND of two arrays element-wise.
`bitwise_or(x1, x2, /[, out, where, casting, …])`	Compute the bit-wise OR of two arrays element-wise.
`bitwise_xor(x1, x2, /[, out, where, …])`	Compute the bit-wise XOR of two arrays element-wise.
`invert(x, /[, out, where, casting, order, …])`	Compute bit-wise inversion, or bit-wise NOT, element-wise.
`left_shift(x1, x2, /[, out, where, casting, …])`	Shift the bits of an integer to the left.
`right_shift(x1, x2, /[, out, where, …])`	Shift the bits of an integer to the right.
Comparison functions[1]	
`greater(x1, x2, /[, out, where, casting, …])`	Return the truth value of (x1 > x2) element-wise.
`greater_equal(x1, x2, /[, out, where, …])`	Return the truth value of (x1 >= x2) element-wise.
`less(x1, x2, /[, out, where, casting, …])`	Return the truth value of (x1 < x2) element-wise.
`less_equal(x1, x2, /[, out, where, casting, …])`	Return the truth value of (x1 =< x2) element-wise.

`not_equal(x1, x2, /[, out, where, casting, …])`	Return (x1 != x2) element-wise.
`equal(x1, x2, /[, out, where, casting, …])`	Return (x1 == x2) element-wise.
`logical_and(x1, x2, /[, out, where, …])`[2]	Compute the truth value of x1 AND x2 element-wise.
`logical_or(x1, x2, /[, out, where, casting, …])`	Compute the truth value of x1 OR x2 element-wise.
`logical_xor(x1, x2, /[, out, where, …])`	Compute the truth value of x1 XOR x2, element-wise.
`logical_not(x, /[, out, where, casting, …])`	Compute the truth value of NOT x element-wise.
`maximum(x1, x2, /[, out, where, casting, …])`	Element-wise maximum of array elements.
`minimum(x1, x2, /[, out, where, casting, …])`	Element-wise minimum of array elements.
`fmax(x1, x2, /[, out, where, casting, …])`	Element-wise maximum of array elements.
`fmin(x1, x2, /[, out, where, casting, …])`	Element-wise minimum of array elements.
Floating functions These all work element-by-element over an array, returning an array output. The description details only a single operation.	
`isfinite(x, /[, out, where, casting, order, …])`	Test element-wise for finiteness (not infinity or not Not a Number).
`isinf(x, /[, out, where, casting, order, …])`	Test element-wise for positive or negative infinity.
`isnan(x, /[, out, where, casting, order, …])`	Test element-wise for NaN and return result as a boolean array.
`isnat(x, /[, out, where, casting, order, …])`	Test element-wise for NaT (not a time) and return result as a boolean array.
`fabs(x, /[, out, where, casting, order, …])`	Compute the absolute values element-wise.
`signbit(x, /[, out, where, casting, order, …])`	Returns element-wise True where signbit is set (less than zero).
`copysign(x1, x2, /[, out, where, casting, …])`	Change the sign of x1 to that of x2, element-wise.
`nextafter(x1, x2, /[, out, where, casting, …])`	Return the next floating-point value after x1 towards x2, element-wise.
`spacing(x, /[, out, where, casting, order, …])`	Return the distance between x and the nearest adjacent number.
`modf(x[, out1, out2], / [[, out, where, …])`	Return the fractional and integral parts of an array, element-wise.
`ldexp(x1, x2, /[, out, where, casting, …])`	Returns x1 * 2**x2, element-wise.

`frexp(x[, out1, out2], / [[, out, where, …])`	Decompose the elements of x into mantissa and twos exponent.
`fmod(x1, x2, /[, out, where, casting, …])`	Return the element-wise remainder of division.
`floor(x, /[, out, where, casting, order, …])`	Return the floor of the input, element-wise.
`ceil(x, /[, out, where, casting, order, …])`	Return the ceiling of the input, element-wise.
`trunc(x, /[, out, where, casting, order, …])`	Return the truncated value of the input, element-wise.

[1]Warning — do not use the Python keywords **and** and **or** to combine logical array expressions. These keywords will test the truth value of the entire array (not element-by-element as you might expect). Use the bitwise operators **&** and **|** instead.

[2]Warning — bit-wise operators **&** and **|** are the proper way to perform element-by-element comparisons. Be sure you understand the operator precedence: `(a > 2) & (a < 5)` instead of `a > 2 & a < 5`.

Vectorizing functions

- Many functions "just work"
- `np.vectorize()` allows user-defined function to be broadcast.

ufuncs will automatically be broadcast across any array to which they are applied. For user-defined functions that don’t correctly broadcast, NumPy provides the **vectorize()** function. It takes a function which accepts one or more scalar values (float, integers, etc.) and returns a single scalar value.

Example

np_vectorize.py

```
import time
import numpy as np

sample_data = np.loadtxt(   # Create some sample data
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
)

def set_default(value, limit, default): # Define function with more than one parameter
    if value > limit:
        value = default

    return value

MAX_VALUE = 50      # Define max value
DEFAULT_VALUE = -1  # Define default value

print("Version 1: looping over arrays")
start = time.perf_counter() # Get the current time as Unix timestamp (large float)
try:
    version1_array = np.zeros(sample_data.shape, dtype=int)  # Create array to hold
results
    for i, row in enumerate(sample_data):  # Iterate over rows and columns of input array
        for j, column in enumerate(row):
            version1_array[i, j] = set_default(sample_data[i, j], MAX_VALUE,
DEFAULT_VALUE)   # Call function and put result in new array
except ValueError as err:
    print("Function failed:", err)
else:
    end = time.perf_counter()  # Get current time
    elapsed = end - start  # Get elapsed number of seconds and print them out
    print(version1_array)
    print("took {:.5f} seconds".format(elapsed))
finally:
    print()


print("Version 2: broadcast without vectorize()")
start = time.perf_counter()
try:
    print("Without sp.vectorize:")
    version2_array = set_default(sample_data, MAX_VALUE, DEFAULT_VALUE) # Pass array to
function; it fails because it has more than one parameter
```

```
except ValueError as err:
    print("Function failed:", err)
else:
    end = time.perf_counter()
    elapsed = end - start
    print(version2_array)
    print("took {:.5f} seconds".format(elapsed))
finally:
    print()

print("Version 3: broadcast with vectorize()")
set_default_vect = np.vectorize(set_default)  # Convert function to vectorized version --
creates function that takes one parameter and has the other two "embedded" in it

start = time.perf_counter()
try:
    print("With sp.vectorize:")
    version3_array = set_default_vect(sample_data, MAX_VALUE, DEFAULT_VALUE) # Call
vectorized version with same parameters
except ValueError as err:
    print("Function failed:", err)
else:
    end = time.perf_counter()
    elapsed = end - start
    print(version3_array)
    print("took {:.5f} seconds".format(elapsed))
finally:
    print()
```

np_vectorize.py

```
Version 1: looping over arrays
[[-1 -1 -1 -1 50  4]
 [40 -1  9 -1 -1 17]
 [18 23  2 -1  1  9]
 ...
 [26 20 -1 46 38 23]
 [ 9  5 -1 23  2 26]
 [46 34 25  8 39 34]]
took 0.00718 seconds

Version 2: broadcast without vectorize()
Without sp.vectorize:
Function failed: The truth value of an array with more than one element is ambiguous. Use
a.any() or a.all()

Version 3: broadcast with vectorize()
With sp.vectorize:
[[-1 -1 -1 -1 50  4]
 [40 -1  9 -1 -1 17]
 [18 23  2 -1  1  9]
 ...
 [26 20 -1 46 38 23]
 [ 9  5 -1 23  2 26]
 [46 34 25  8 39 34]]
took 0.00312 seconds
```

Getting help

- Several help functions
  - `numpy.info()`
  - `numpy.lookfor()`
  - `numpy.source()`

NumPy has several functions for getting help. The first is `numpy.info()`, which provides a brief explanation of a function, class, module, or other object as well as some code examples.

If you're not sure what function you need, you can try `numpy.lookfor()`, which does a keyword search through the NumPy documentation.

These functions are convenient when using **iPython** or **Jupyter**.

Example

np_info.py

```
import numpy as np
import scipy.fftpack as ff


def main():
    np.info(ff.fft)  # Get help on the fft() function

    print('-' * 60)

    np.source(ff.fft)  # View the source of the fft() function

    print('-' * 60)

    np.lookfor('convolve') # search np docs


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

Iterating

- Similar to normal Python
- Iterates through first dimension
- Use array.flat to iterate through all elements
- Don’t do it unless you have to

Iterating through a NumPy array is similar to iterating through any Python list; iteration is across the first dimension. Slicing and indexing can be used.

To iterate across every element, use `array.flat`.

However, iterating over a NumPy array is generally much less efficient than using a *vectorized* approach — calling a *ufunc* or directly applying a math operator. Some tasks may require it, but you should avoid it if possible.

Example

np_iterating.py

```
import numpy as np

a = np.array(
    [[70, 31, 21, 76],
     [23, 29, 71, 12]]
)  # sample array

print('a =>\n', a)
print()

print("for row in a: =>")
for row in a:  # iterate over rows
    print("row:", row)
print()

print("for column in a.T:")
for column in a.T:  # iterate over columns by transposing the array
    print("column:", column)
print()

print("for elem in a.flat: =>")
for elem in a.flat:  # iterate over all elements (row-major)
    print("element:", elem)
```

np_iterating.py

```
a =>
 [[70 31 21 76]
 [23 29 71 12]]

for row in a: =>
row: [70 31 21 76]
row: [23 29 71 12]

for column in a.T:
column: [70 23]
column: [31 29]
column: [21 71]
column: [76 12]

for elem in a.flat: =>
element: 70
element: 31
element: 21
element: 76
element: 23
element: 29
element: 71
element: 12
```

Matrix Multiplication

- Use normal ndarrays
- Most operations same as ndarray
- Use @ for multiplication

For traditional matrix operations, use a normal ndarray. Most operations are the same as for ndarrays. For matrix (diagonal) multiplication, use the @ (matrix multiplication) operator.

For transposing, use *array*.transpose(), or just *array*.T.

NOTE There was formerly a Matrix type in NumPy, but it is deprecated since the addition of the @ operator in Python 3.5

Example

np_matrices.py

```
import numpy as np

m1 = np.array(
    [[2, 4, 6],
     [10, 20, 30]]
)  # sample 2x3 array

m2 = np.array([[1, 15],
               [3, 25],
               [5, 35]])  # sample 3x2 array

print('m1 =>\n', m1)
print()

print('m2 =>\n', m2)
print()

print('m1 * 10 =>\n', m1 * 10)  # multiply every element of m1 times 10
print()

print('m1 @ m2 =>\n', m1 @ m2)  # matrix multiply m1 times m2 -- diagonal product
print()
```

np_matrices.py

```
m1 =>
 [[ 2  4  6]
 [10 20 30]]

m2 =>
 [[ 1 15]
 [ 3 25]
 [ 5 35]]

m1 * 10 =>
 [[ 20  40  60]
 [100 200 300]]

m1 @ m2 =>
 [[  44  340]
 [ 220 1700]]
```

Data Types

- Default is **float**
- Data type is inferred from initialization data
- Can be specified with `arange()`, `ones()`, `zeros()`, etc.

Numpy defines around 30 numeric data types. Integers can have different sizes and byte orders, and be either signed or unsigned. The data type is normally inferred from the initialization data. When using `arange()`, `ones()`, etc., to create arrays, the **dtype** parameter can be used to specify the data type.

The default data type is **np.float_**, which maps to the Python builtin type **float**.

The data type cannot be changed after an array is created.

See https://numpy.org/devdocs/user/basics.types.html for more details.

Example

np_data_types.py

```
import numpy as np

r1 = np.arange(45)  # create array -- arange() defaults to int
r1.shape = (3, 3, 5)  # create array -- passing float makes all elements float
print('r1 datatype:', r1.dtype)
print('r1 =>\n', r1, '\n')

r2 = np.arange(45.)  # create array -- set datatype to short int
r2.shape = (3, 3, 5)
print('r2 datatype:', r2.dtype)
print('r2 =>\n', r2, '\n')

r3 = np.arange(45, dtype=np.int16)  # create array -- set datatype to short int
r3.shape = (3, 3, 5)
print('r3 datatype:', r3.dtype)
print('r3 =>\n', r3, '\n')
```

np_data_types.py

```
r1 datatype: int64
r1 =>
 [[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]

 [[30 31 32 33 34]
  [35 36 37 38 39]
  [40 41 42 43 44]]]

r2 datatype: float64
r2 =>
 [[[ 0.  1.  2.  3.  4.]
  [ 5.  6.  7.  8.  9.]
  [10. 11. 12. 13. 14.]]

 [[15. 16. 17. 18. 19.]
  [20. 21. 22. 23. 24.]
  [25. 26. 27. 28. 29.]]

 [[30. 31. 32. 33. 34.]
  [35. 36. 37. 38. 39.]
  [40. 41. 42. 43. 44.]]]

r3 datatype: int16
r3 =>
 [[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]

 [[30 31 32 33 34]
  [35 36 37 38 39]
  [40 41 42 43 44]]]
```

Reading and writing Data

- Read data from files into **ndarray**
- Text files
  - `loadtxt()`
  - `savetxt()`
  - `genfromtxt()`
- Binary (or text) files
  - `fromfile()`
  - `tofile()`

NumPy has several functions for reading data into an array.

`numpy.loadtxt()` reads a delimited text file. There are many options for fine-tuning the import.

`numpy.genfromtxt()` is similar to `numpy.loadtxt()`, but also adds support for handling missing data

Both functions allow skipping rows, user-defined per-column converters, setting the data type, and many others.

To save an array as a text file, use the `numpy.savetxt()` function. You can specify delimiters, header, footer, and formatting.

To read binary data, use `numpy.fromfile()`. It expects a file to contain all the same data type, i.e., ints or floats of a specified type. It will default to floats. `fromfile()` can also be used to read text files.

To save as binary data, you can use `numpy.tofile()`, but **tofile()** and **fromfile()** are not platform-independent. See the next section on **save()** and **load()** for platform-independent I/O.

Example

np_savetxt_loadtxt.py

```
import numpy as np

sample_data = np.loadtxt(   # Load data from space-delimited file
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
    dtype=float
)

print(sample_data)
print('-' * 60)

sample_data  /= 10  # Modify sample data

float_file_name = 'save_data_float.txt'

np.savetxt(float_file_name, sample_data, delimiter=",", fmt="%5.2f")  # Write data to
text file as floats, rounded to two decimal places, using commas as delimiter

int_file_name = 'save_data_int.txt'

np.savetxt(int_file_name, sample_data, delimiter=",", fmt="%d")  # Write data to text
file as ints, using commas as delimiter

data = np.loadtxt(float_file_name, delimiter=",")  # Read data back into ndarray
print(data)
```

np_savetxt_loadtxt.py

```
[[63. 51. 59. 61. 50.  4.]
 [40. 66.  9. 64. 63. 17.]
 [18. 23.  2. 61.  1.  9.]
 ...
 [26. 20. 54. 46. 38. 23.]
 [ 9.  5. 59. 23.  2. 26.]
 [46. 34. 25.  8. 39. 34.]]
------------------------------------------------------------
[[6.3 5.1 5.9 6.1 5.  0.4]
 [4.  6.6 0.9 6.4 6.3 1.7]
 [1.8 2.3 0.2 6.1 0.1 0.9]
 ...
 [2.6 2.  5.4 4.6 3.8 2.3]
 [0.9 0.5 5.9 2.3 0.2 2.6]
 [4.6 3.4 2.5 0.8 3.9 3.4]]
```

Example

np_tofile_fromfile.py

```python
import numpy as np

sample_data = np.loadtxt(   # Read in sample data
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
    dtype=float
)

sample_data  /= 10  # Modify sample data

print(sample_data)
print("-" * 60)

file_name = 'sample.dat'

sample_data.tofile(file_name)  # Write data to file (binary, but not portable)

data = np.fromfile(file_name)  # Read binary data from file as one-dimensional array
data.shape = sample_data.shape  # Set shape to shape of original array

print(data)
```

np_tofile_fromfile.py

```
[[6.3 5.1 5.9 6.1 5.  0.4]
 [4.  6.6 0.9 6.4 6.3 1.7]
 [1.8 2.3 0.2 6.1 0.1 0.9]
 ...
 [2.6 2.  5.4 4.6 3.8 2.3]
 [0.9 0.5 5.9 2.3 0.2 2.6]
 [4.6 3.4 2.5 0.8 3.9 3.4]]
------------------------------------------------------------
[[6.3 5.1 5.9 6.1 5.  0.4]
 [4.  6.6 0.9 6.4 6.3 1.7]
 [1.8 2.3 0.2 6.1 0.1 0.9]
 ...
 [2.6 2.  5.4 4.6 3.8 2.3]
 [0.9 0.5 5.9 2.3 0.2 2.6]
 [4.6 3.4 2.5 0.8 3.9 3.4]]
```

Saving and retrieving arrays

- Efficient binary format
- Save as NumPy data
    - Use `numpy.save()`
- Read into ndarray
    - Use `numpy.load()`

To save an array as a NumPy data file, use `numpy.save()`. This will write the data out to a specified file name, adding the extension '.npy'.

To read the data back into a NumPy ndarray, use `numpy.load()`. Data are read and written in a way that preserves precision and endianness.

This the most efficient way to store numeric data for later retrieval, compared to **savetext()** and **loadtext()** or **tofile()** and **fromfile()**. Files written with `numpy.save()` are not human-readable.

Example

np_save_load.py

```python
import numpy as np

sample_data = np.loadtxt(   # Read some sample data into an ndarray
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
    dtype=int
)

sample_data  *= 100  # Modify the sample data (multiply every element by 100)

print(sample_data)

file_name = 'sampledata'

np.save(file_name, sample_data)  # Write entire array out to NumPy-format data file (adds
.npy extension)

retrieved_data = np.load(file_name + '.npy')  # Retrieve data from saved file

print('-' * 60)
print(retrieved_data)
```

np_save_load.py

```
[[6300 5100 5900 6100 5000  400]
 [4000 6600  900 6400 6300 1700]
 [1800 2300  200 6100  100  900]
 ...
 [2600 2000 5400 4600 3800 2300]
 [ 900  500 5900 2300  200 2600]
 [4600 3400 2500  800 3900 3400]]
------------------------------------------------------------
[[6300 5100 5900 6100 5000  400]
 [4000 6600  900 6400 6300 1700]
 [1800 2300  200 6100  100  900]
 ...
 [2600 2000 5400 4600 3800 2300]
 [ 900  500 5900 2300  200 2600]
 [4600 3400 2500  800 3900 3400]]
```

Chapter 14 Exercises

Exercise 14-1 (big_arrays.py)

Starting with the file big_arrays.py, convert the Python list values into a NumPy array.

Make a copy of the array named values_x_3 with all values multiplied by 3.

Print out values_x_3

Exercise 14-2 (create_range.py)

Using arange(), create an array of 35 elements.

Reshape the arrray to be 5 x 7 and print it out.

Reshape the array to be 7 x 5 and print it out.

Exercise 14-3 (create_linear_space.py)

Using linspace(), create an array of 500 elements evenly spaced between 100 and 200.

Reshape the array into 5 x 10 x 10.

Multiply every element by .5

Print the result.

Chapter 15: SciPy Overview

Objectives

- Understand the motivation for scipy
- See how scipy fits into the Python Scientific stack
- Tour scipy’s many subpackages
- Import scipy and friends using standard abbreviations
- Learn how scipy integrates numpy
- Use the scipy documentation commands

About scipy

- Part of the "Python Scientific Stack"
- Often used with matplotlib
- Many mathematical and statistical algorithms
- Includes numpy "under the hood"
- Can be used with iPython
- Very large collection of routines and subpackages

scipy is a collection of modules and submodules for doing scientific (mostly numerical) analysis.

The scipy module itself acts as an umbrella module, or repository, for many useful submodules.

Although it can be used alone, numpy is part of scipy, and many useful numpy functions are aliased into the scipy namespace.

In addition, many common functions from scipy’s dozens of submodules have been aliased to the scipy namespace.

All of the top-level **numpy** functions are available through scipy as well.

Polynomials

- Create a poly1d object
- Represented in either of two ways
  - list of coefficients (1st element is coefficient of highest power)
  - list of roots
- Call polynomial with value to solve for
- r attribute represents list of roots

Polynomials can be represented in scipy in two ways using numpy's poly1d() method, which takes a list of coefficients; the other is to just provide a list of coefficients, where the first element is the coefficient of the highest power.

The poly1d() method takes a list of coefficients (or roots) and returns a poly1d object.

Treating the polynomial object like a string returns a text representation of the polynomial.

By default, pass an iterable of integers to poly1d which represent the coefficients. To specify roots, pass a second parameter to poly1d with a true value.

The variable used when displaying the polynomial is normally x. To use a different variable, add a third parameter with a string.

To solve for a specific value, call the polynomial with that value. The r property of the polynomial contains the roots.

You can use the addition, subtraction, division, multiplication, and exponentiation operators between polynomials and scalar values.

NOTE poly1d is automatically imported to scipy's namespace as well

Example

sp_polynomials.py

```python
import scipy as sp

p1 = sp.poly1d([2, 1, 4])  # 2,1,4 are coefficients
print(p1, '\n')

print(p1(.75), '\n')  # evaluate for x = .75

print(p1.r, '\n')  # get the roots

p2 = sp.poly1d([2, 1, -4], True)  # 2,1,-4 are roots
print(p2, '\n')

print(p2(.75))  # evaluate for x = .75
print(p2.r), '\n'  # get the roots

p3 = sp.poly1d([1, 2, 3], False, 'm')  # 1,2,3 are coefficients, variable is 'm'
print(p3, '\n')

print(p3(100), '\n')  # evaluate for m = 100

print(p3.r, '\n')  # get the roots

p4 = sp.poly1d([1, 2])  # polynomial arithmetic
p5 = sp.poly1d([3, 4], '\n')

print(p4, '\n')

print(p5, '\n')

print(p4 + p5, '\n')

print(p4 - p5, '\n')

print(p4 ** 3, '\n')
```

sp_polynomials.py

```
   2
2 x + 1 x + 4

5.875

[-0.25+1.39194109j -0.25-1.39194109j]

   3     2
1 x + 1 x - 10 x + 8

1.484375
[-4.  2.  1.]
   2
1 m + 2 m + 3

10203

[-1.+1.41421356j -1.-1.41421356j]


1 x + 2

   2
1 x - 7 x + 12

   2
1 x - 6 x + 14

    2
-1 x + 8 x - 10

   3     2
1 x + 6 x + 12 x + 8
```

Working with SciPy

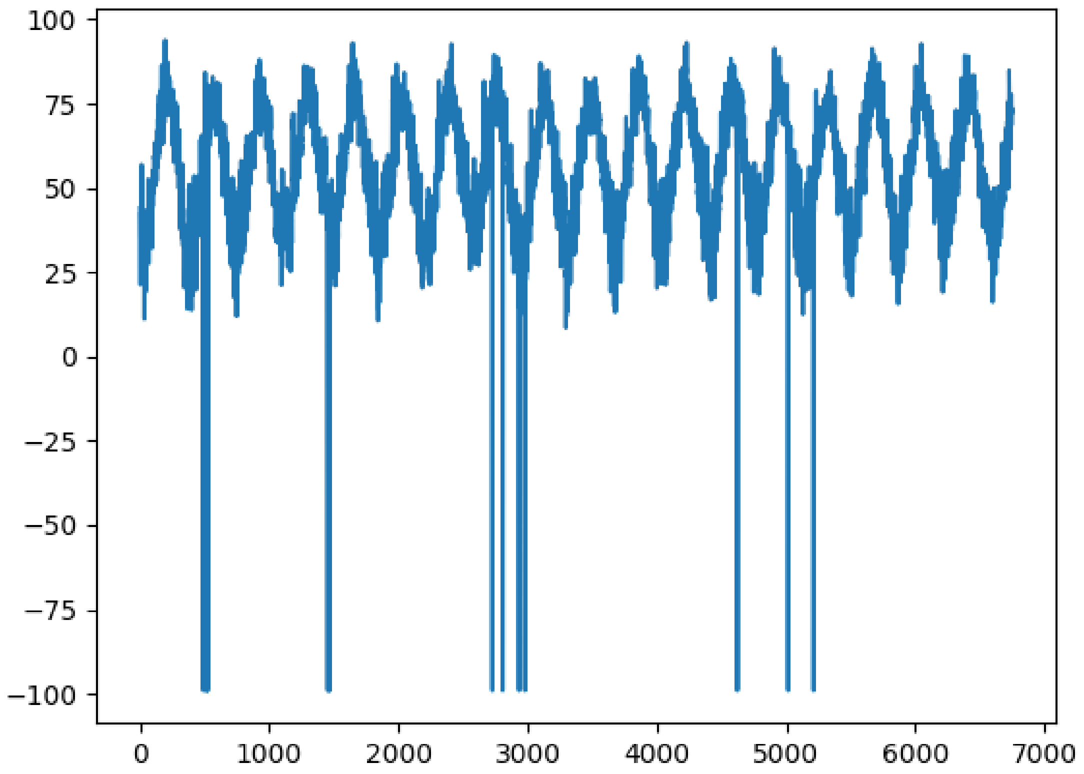
- Some functions imported to scipy
- Hundreds more in subpackages
- Most functions have similar interfaces
- Use numpy plus scipy.subpackage routines as needed

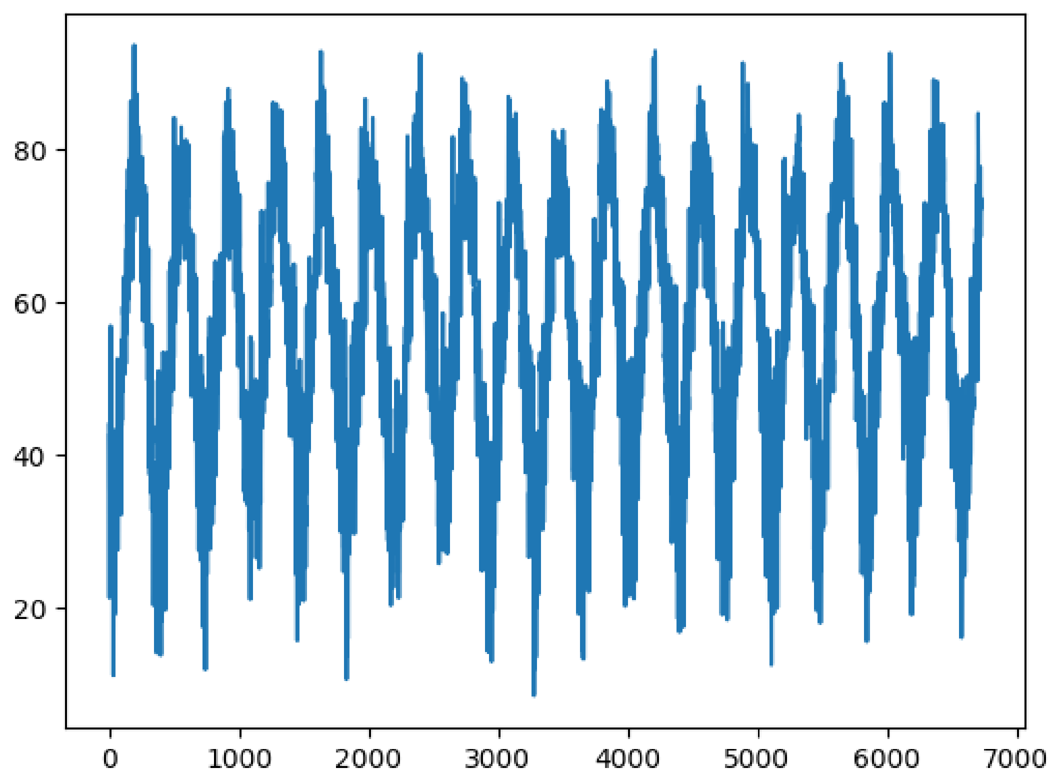
scipy’s subpackages have hundreds of functions. For convenience, some of them have been imported into the scipy namespace. While numpy is mostly about arrays and matrices, there are some useful data handling functions as well.

Example

sp_functions.py

```
import numpy as np
import matplotlib.pyplot as plt

dt = np.dtype([('Month', 'int8'), ('Day', 'int8'), ('Year', 'int16'), ('Temp',
'float64')])  # define custom numpy datatype
data = np.loadtxt('../DATA/weather/NYNEWYOR.txt', dtype=dt)  # read flat-file data into
numpy array

print(data['Temp'])

temps = data['Temp']  # get data from 'Temp' column

plt.plot(temps)  # plot days against temps
plt.show()  # display plot
plt.cla()  # clear first axis but not the whole figure

normalized_data = data[data['Temp'] > -50]  # remove readings < -50, which seem to be
default N/A values
temps = normalized_data['Temp']  # grab temps again
plt.plot(temps)  # replot
plt.show()
```


100
75
50
25
0
−25
−50
−75
−100
0
1000
2000
3000
4000
5000
6000
7000


80
60
40
20
0
1000
2000
3000
4000
5000
6000
7000

SciPy Subpackages

- cluster — Clustering algorithms
- constants — Physical and mathematical constants
- fftpack --Fast Fourier Transform routines
- integrate — Integration and ordinary differential equation solvers
- interpolate — Interpolation and smoothing splines
- io — Input and Output
- linalg — Linear algebra
- ndimage — N-dimensional image processing
- odr — Orthogonal distance regression
- optimize — Optimization and root-finding routines
- signal — Signal processing
- sparse — Sparse matrices and associated routines
- spatial — Spatial data structures and algorithms
- special — Special functions
- stats — Statistical distributions and functions
- weave — C/C++ integration

Python and C/C++

- Ctypes allows loading of shared libraries (dll/dylib/so)
- Cython is optimizing static compiler

When there is not a fast numpy/scipy routine that implements a needed algorithm, you can write the algorithm in C/C++.

This may not help with code that already uses existing numpy/scipy functions, but if the code contains nested loops, the speedup can be significant.

Of course, extending scipy with C requires a C development package. This can be either **Microsoft Visual C++** or **MinGW** on Windows, and is typically **gcc** on other platforms.

The simplest approach is **ctypes**. This is part of the standard library, and allows direct import of C/C shared libraries (.dll, .dylib, or .so) without writing any custom C/C code.

cython is a version of Python that automatically pre-compiles selected Python code into C. It looks like Python, but has type declarations similar to those in C.

numba is a library that contains the **jit** decorator. Using this will dramatically speed up calculation-heavy functions without changing any code.

Tour of SciPy subpackages

- 20 subpackages
- Hundreds of routines

SciPy has 20 main subpackages, each of which can have hundreds of routines. They are all designed to work with NumPy arrays, and there is some overlap among them. Some of the routines are also available from the **scipy** package itself, without having to import the subpackage.

We will now visit the SciPy documentation for a brief tour of the SciPy packages, at https://docs.scipy.org/doc/scipy/reference/

Chapter 16: Introduction to Pandas

Objectives

- Understand what the pandas module provides
- Load data from CSV and other files
- Access data tables
- Extract rows and columns using conditions
- Calculate statistics for rows or columns

About pandas

- Reads data from file, database, or other sources
- Deals with real-life issues such as invalid data
- Powerful selecting and indexing tools
- Builtin statistical functions
- Munge, clean, analyze, and model data
- Works with numpy and matplotlib

pandas is a package designed to make it easy to get, organize, and analyze large datasets. Its strengths lie in its ability to read from many different data sources, and to deal with real-life issues, such as missing, incomplete, or invalid data.

pandas also contains functions for calculating means, sums and other kinds of analysis.

For selecting desired data, pandas has many ways to select and filter rows and columns.

It is easy to integrate pandas with NumPy, Matplotlib, and other scientific packages.

While pandas can handle three (or higher) dimensional data, it is generally used with two-dimensional (row/column) data, which can be visualized like a spreadsheet.

pandas provides powerful split-apply-combine operations — **groupby** enables transformations, aggregations, and easy-access to plotting functions. It is easy to emulate R's plyr package via pandas.

NOTE pandas gets its name from *pan*el *da*ta *s*ystem

Tidy data

- Tidy data is neatly grouped
- Data
  - *Value* = "observation"
  - *Column* = "variable"
  - *Row* = "related observations"
- Pandas best with tidy data

A dataset contains *values*. Those values can be either numbers or strings. Values are grouped into *variables*, which are usually represented as *columns*. For instance, a column might contain "unit price" or "percentage of NaCL". A group of related values is called an *observation*. A *row* represents an observation. Every combination of row and column is a single value.

When data is arranged this way, it is said to be "tidy". Pandas is designed to work best with tidy data.

For instance,

```
Product    SalesYTD
oranges    5000
bananas    1000
grapefruit 10000
```

is tidy data. The variables are "Product" and "SalesYTD", and the observations are the names of the fruits and the sales figures.

The following dataset is NOT tidy:

```
Fruit     oranges bananas grapefruit
SalesYTD  5000    1000    10000
```

To make selecting data easy, Pandas dataframes always have variable labels (columns) and observation labels (row indexes). A row index could be something simple like increasing integers, but it could also be a time series, or any set of strings, including a column pulled from the data set.

TIP variables could be called "features" and observations could be called "samples"

NOTE See https://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html for a detailed discussion of tidy data.

pandas architecture

- Two main structures: Series and DataFrame
- Series – one-dimensional
- DataFrame – two-dimensional

The two main data structures in pandas are the **Series** and the **DataFrame**. A series is a one-dimensional indexed list of values, something like an ordered dictionary. A DataFrame is is a two-dimensional grid, with both row and column indexes (like the rows and columns of a spreadsheet, but more flexible).

You can specify the indexes, or pandas will use successive integers. Each row or column of a DataFrame is a Series.

NOTE pandas used to support the **Panel** type, which is more more or less a collection of DataFrames, but Panel has been deprecated in favor of hierarchical indexing.

Series

- Indexed list of values
- Similar to a dictionary, but ordered
- Can get sum(), mean(), etc.
- Use index to get individual values
- indexes are not positional

A Series is an indexed sequence of values. Each item in the sequence has an index. The default index is a set of increasing integer values, but any set of values can be used.

For example, you can create a series with the values 5, 10, and 15 as follows:

```
s1 = pd.Series([5,10,15])
```

This will create a Series indexed by [0, 1, 2]. To provide index values, add a second list:

```
s2 = pd.Series([5,10,15], ['a','b','c'])
```

This specifies the indexes as 'a', 'b', and 'c'.

You can also create a Series from a dictionary. pandas will put the index values in order:

```
s3 = pd.Series({'b':10, 'a':5, 'c':15})
```

There are many methods that can be called on a Series, and Series can be indexed in many flexible ways.

Example

pandas_series.py

```python
from numpy.random import default_rng
import pandas as pd

NUM_DATA_POINTS = 10
index = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

rng = default_rng()
data = rng.standard_normal(NUM_DATA_POINTS)

s1 = pd.Series(data, index=index)  # create series with specified index
s2 = pd.Series(data)  # create series with auto-generated index (0, 1, 2, 3, ...)

print("s1:", s1, "\n")
print("s2:", s2, "\n")

print("selecting elements")
print(s1[['h', 'b']], "\n")  # select items from series

print(s1[['a', 'b', 'c']], "\n")  # select items from series

print("slice of elements")
print(s1['b':'d'], "\n")  # select slice of elements

print("sum(), mean(), min(), max():")
print(s1.sum(), s1.mean(), s1.min(), s1.max(), "\n")  # get stats on series

print("cumsum(), cumprod():")
print(s1.cumsum(), s1.cumprod(), "\n")  # get stats on series

print('a' in s1)  # test for existence of label
print('m' in s1)  # test for existence of label
print()

s3 = s1 * 10  # create new series with every element of s1 multiplied by 10
print("s3 (which is s1 * 10)")
print(s3, "\n")

s1['e'] *= 5

print("boolean mask where s3 > 0:")
print(s3 > 0, "\n")  # create boolean mask from series

print("assign -1 where mask is true")
```

```
s3[s3 < 5] = -1  # set element to -1 where mask is True
print(s3, "\n")

s4 = pd.Series([-0.204708, 0.478943, -0.519439])  # create new series
print("s4.max(), .min(), etc.")
print(s4.max(), s4.min(), s4.max() - s4.min(), '\n')  # print stats

s = pd.Series([5, 10, 15], ['a', 'b', 'c'])  # create new series with index
print("creating series with index")
print(s)
```

pandas_series.py

```
s1: a    0.223417
b   -1.341152
c   -0.140739
d   -1.024818
e    0.432619
f    0.679758
g   -0.376235
h    0.498322
i   -0.717258
j    0.462191
dtype: float64

s2: 0    0.223417
1   -1.341152
2   -0.140739
3   -1.024818
4    0.432619
5    0.679758
6   -0.376235
7    0.498322
8   -0.717258
9    0.462191
dtype: float64

selecting elements
h    0.498322
b   -1.341152
dtype: float64

a    0.223417
b   -1.341152
c   -0.140739
dtype: float64

slice of elements
b   -1.341152
c   -0.140739
d   -1.024818
dtype: float64

sum(), mean(), min(), max():
-1.3038946649945105 -0.13038946649945105 -1.3411517337557615 0.6797582068040133

cumsum(), cumprod():
a    0.223417
```

```
b   -1.117735
c   -1.258474
d   -2.283291
e   -1.850672
f   -1.170914
g   -1.547149
h   -1.048828
i   -1.766086
j   -1.303895
dtype: float64 a    0.223417
b   -0.299636
c    0.042170
d   -0.043217
e   -0.018696
f   -0.012709
g    0.004782
h    0.002383
i   -0.001709
j   -0.000790
dtype: float64

True
False

s3 (which is s1 * 10)
a     2.234167
b   -13.411517
c    -1.407389
d   -10.248175
e     4.326190
f     6.797582
g    -3.762351
h     4.983216
i    -7.172579
j     4.621910
dtype: float64

boolean mask where s3 > 0:
a     True
b    False
c    False
d    False
e     True
f     True
g    False
h     True
i    False
j     True
```

```
dtype: bool

assign -1 where mask is true
a   -1.000000
b   -1.000000
c   -1.000000
d   -1.000000
e   -1.000000
f    6.797582
g   -1.000000
h   -1.000000
i   -1.000000
j   -1.000000
dtype: float64

s4.max(), .min(), etc.
0.478943 -0.519439 0.998382

creating series with index
a     5
b    10
c    15
dtype: int64
```

DataFrames

- Two-dimensional grid of values
- Row and column labels (indexes)
- Rich set of methods
- Powerful indexing

A DataFrame is the workhorse of pandas. It represents a two-dimensional grid of values, containing indexed rows and columns, something like a spreadsheet.

There are many ways to create a DataFrame. They can be modified to add or remove rows/columns. Missing or invalid data can be eliminated or normalized.

DataFrames can be initialized from many kinds of data. See the table on the next page for a list of possibilities.

NOTE The panda DataFrame is modeled after R’s data.frame

Table 20. DataFrame Initializers

Initializer	Description
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the “dict of arrays” case
dict of Series	Each value becomes a column. Indexes from each Series are union-ed together to form the result’s row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column. Keys are union-ed to form the row index as in the “dict of Series” case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame’s column labels
List of lists or tuples	Treated as the “2D ndarray” case
Another DataFrame	The DataFrame’s indexes are used unless different ones are passed
NumPy MaskedArray	Like the “2D ndarray” case except masked values become NA/missing in the DataFrame result

IMPORTANT Most, if not all, of the time you will create Series and Dataframes by reading data.

Example

pandas_simple_dataframe.py

```
import pandas as pd
from printheader import print_header


cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  # column names
indices = ['a', 'b', 'c', 'd', 'e', 'f']  # row names

values = [  # sample data
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]
print_header('cols')
print(cols, '\n')

print_header('indices')
print(indices, '\n')

print_header('values')
print(values, '\n')

df = pd.DataFrame(values, index=indices, columns=cols)  # create dataframe with row and
column names
print_header('DataFrame df')
print(df, '\n')

print_header("df['gamma']")
print(df['gamma'])  # select column 'gamma'
```

pandas_simple_dataframe.py

```
==================================================
=                      cols                      =
==================================================
['alpha', 'beta', 'gamma', 'delta', 'epsilon']

==================================================
=                    indices                     =
==================================================
['a', 'b', 'c', 'd', 'e', 'f']

==================================================
=                     values                     =
==================================================
[[100, 110, 120, 130, 140], [200, 210, 220, 230, 240], [300, 310, 320, 330, 340], [400,
410, 420, 430, 440], [500, 510, 520, 530, 540], [600, 610, 620, 630, 640]]

==================================================
=                  DataFrame df                  =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

==================================================
=                  df['gamma']                   =
==================================================
a    120
b    220
c    320
d    420
e    520
f    620
Name: gamma, dtype: int64
```

Reading Data

- Supports many data formats
- Reads headings to create column indexes
- Auto-creates indexes as needed
- Can used specified column as row index

Pandas supports many different input formats. It will read file headings and use them to create column indexes. By default, it will use integers for row indexes, but you can specify a column to use as the index, or provide a list of index values.

The **read_…()** functions have many options for controlling and parsing input. For instance, if large integers in the file contain commas, the thousands options let you set the separator as comma (in the US), so it will ignore them.

read_csv() is the most frequently used function, and has many options. It can also be used to read generic flat-file formats. **read_table** is similar to **read_csv()**, but doesn't assume CSV format.

There are corresponding **to_…()** functions for many of the read functions. `to_csv()` and `to_ndarray()` are very useful.

NOTE

See **Jupyter** notebook **pandas_Input_Demo** (in the **NOTEBOOKS** folder) for examples of reading most types of input.

See https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html?highlight=output#io-html for details on the I/O functions.

Example

pandas_read_csv.py

```
import pandas as pd

df = pd.read_csv('../DATA/sales_records.csv')  # Read CSV data into dataframe. Pandas
automatically uses the first row as column names

print(df.describe())  # Get statistics on the numeric columns (use
`df.describe(include='O')` for text columns)
print()

print(df.info())  # Get information on all the columns ('object' means text/string)
print()

print(df.head(5))  # Display first 5 rows of the dataframe (`df.describe(__n__)` displays
n rows)

df['total_sales'] = df['Units Sold'] * df['Unit Price']
print(df)

print(df.info())
print(df.describe())
```

pandas_read_csv.py

```
           Order ID   Units Sold   Unit Price    Unit Cost
count  5.000000e+03  5000.000000  5000.000000  5000.000000
mean   5.486447e+08  5030.698200   265.745564   187.494144
std    2.594671e+08  2914.515427   218.716695   176.416280
min    1.000909e+08     2.000000     9.330000     6.920000
25%    3.201042e+08  2453.000000    81.730000    35.840000
50%    5.523150e+08  5123.000000   154.060000    97.440000
75%    7.687709e+08  7576.250000   437.200000   263.330000
max    9.998797e+08  9999.000000   668.270000   524.960000

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Region          5000 non-null   object
 1   Country         5000 non-null   object
 2   Item Type       5000 non-null   object
 3   Sales Channel   5000 non-null   object
 4   Order Priority  5000 non-null   object
 5   Order Date      5000 non-null   object
 6   Order ID        5000 non-null   int64
 7   Ship Date       5000 non-null   object
 8   Units Sold      5000 non-null   int64
 9   Unit Price      5000 non-null   float64
 10  Unit Cost       5000 non-null   float64
dtypes: float64(2), int64(2), object(7)
memory usage: 429.8+ KB
None

                           Region  ... Unit Cost
0  Central America and the Caribbean  ...    159.42
1  Central America and the Caribbean  ...     97.44
2                            Europe  ...     31.79
3                              Asia  ...    117.11
4                              Asia  ...     97.44

[5 rows x 11 columns]
                              Region  ... total_sales
0     Central America and the Caribbean  ...   140914.56
1     Central America and the Caribbean  ...   330640.86
2                                Europe  ...   226716.10
3                                  Asia  ...  1854591.20
4                                  Asia  ...  1150758.36
...                                 ...  ...         ...
4995              Australia and Oceania  ...  3545172.35
4996       Middle East and North Africa  ...   117694.56
4997                               Asia  ...  1328477.12
4998                             Europe  ...  1028324.80
4999                 Sub-Saharan Africa  ...   377447.00

[5000 rows x 12 columns]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 12 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Region          5000 non-null   object
 1   Country         5000 non-null   object
 2   Item Type       5000 non-null   object
 3   Sales Channel   5000 non-null   object
 4   Order Priority  5000 non-null   object
 5   Order Date      5000 non-null   object
 6   Order ID        5000 non-null   int64
 7   Ship Date       5000 non-null   object
 8   Units Sold      5000 non-null   int64
 9   Unit Price      5000 non-null   float64
 10  Unit Cost       5000 non-null   float64
 11  total_sales     5000 non-null   float64
dtypes: float64(3), int64(2), object(7)
memory usage: 468.9+ KB
None
           Order ID   Units Sold   Unit Price    Unit Cost   total_sales
count  5.000000e+03  5000.000000  5000.000000  5000.000000  5.000000e+03
mean   5.486447e+08  5030.698200   265.745564   187.494144  1.325738e+06
std    2.594671e+08  2914.515427   218.716695   176.416280  1.475375e+06
min    1.000909e+08     2.000000     9.330000     6.920000  6.531000e+01
25%    3.201042e+08  2453.000000    81.730000    35.840000  2.574168e+05
50%    5.523150e+08  5123.000000   154.060000    97.440000  7.794095e+05
75%    7.687709e+08  7576.250000   437.200000   263.330000  1.839975e+06
max    9.998797e+08  9999.000000   668.270000   524.960000  6.672676e+06
```

Table 21. pandas I/O functions

Format	Input function	Output function
CSV	`read_csv()`	`to_csv()`
Delimited file (generic)	`read_table()`	`to_csv()`
Excel worksheet	`read_excel()`	`to_excel()`
File with fixed-width fields	`read_fwf()`	
Google BigQuery	`read_gbq()`	`to_gbq()`
HDF5	`read_hdf()`	`to_hdf()`
HTML table	`read_html()`	`to_html()`
JSON	`read_json()`	`to_json()`
OS clipboard data	`read_clipboard()`	`to_clipboard()`
Parquet	`read_parquet()`	`to_parquet()`
pickle	`read_pickle()`	`to_pickle()`
SAS	`read_sas()`	
SQL query	`read_sql()`	`to_sql()`

NOTE All **read_...()** functions return a new **DataFrame**, except **read_html()**, which returns a list of **DataFrames**

Data summaries

- `describe()` *basic statistical details*
- `info()` *per-column details (shallow memory use)*
- `info(memory_usage='deep')` *actual memory use*

You can call the `describe()` and `info()` methods on a dataframe to get summaries of the kind of data contained.

The `describe()` method, by default, shows statistics on all numeric columns. Add `include='int'` or `include='float'` to restrict the output to those types. `include='all'` will show all types, including "objects" (AKA text).

To show just objects (strings), use `include='O'`. This will show all text columns. You can compare the **count** and **unique** values to check the *cardinality* of the column, or how many distinct values there are. Columns with few unique values are said to have low cardinality, and are candidates for saving space by using the `Categorical` data type.

The `info()` method will show the names and types of each column, as well as the count of non-null values. Adding `memory_usage='deep'` will display the total memory actually used by the dataframe. (Otherwise, it's only the memory used by the top-level data structures).

Example

pandas_data_summaries.py

```python
import pandas as pd
from printheader import print_header

df = pd.read_csv('../DATA/airport_boardings.csv', thousands=',', index_col=1)

print_header('df.head()')
print(df.head())
print()

print_header('df.describe()')
print(df.describe())

print_header("df.describe(include='int')")
print(df.describe(include='int'))

print_header("df.describe(include='all')")
print(df.describe(include='all'))

print_header("df.info()")
print(df.info())
```

pandas_data_summaries.py

```
==================================================
=                   df.head()                    =
==================================================
                                               Airport  ...  Percent change 2010-2011
Code                                                    ...
ATL   Atlanta, GA (Hartsfield-Jackson Atlanta Intern...  ...                     -22.6
ORD          Chicago, IL (Chicago O'Hare International)  ...                     -25.5
DFW        Dallas, TX (Dallas/Fort Worth International)  ...                     -23.7
DEN                   Denver, CO (Denver International)  ...                     -23.1
LAX          Los Angeles, CA (Los Angeles International)  ...                     -19.6

[5 rows x 9 columns]

==================================================
=                 df.describe()                  =
==================================================
       2001 Rank  ...  Percent change 2010-2011
count  50.000000  ...                 50.000000
mean   26.460000  ...                -23.758000
```

```
std    15.761242  ...                2.435963
min     1.000000  ...              -32.200000
25%    13.250000  ...              -25.275000
50%    26.500000  ...              -23.650000
75%    38.750000  ...              -22.075000
max    59.000000  ...              -19.500000

[8 rows x 8 columns]
==================================================
=           df.describe(include='int')           =
==================================================
       2001 Rank    2001 Total  ...  2011 Rank         Total
count  50.000000  5.000000e+01  ...   50.00000  5.000000e+01
mean   26.460000  9.848488e+06  ...   25.50000  8.558513e+06
std    15.761242  7.042127e+06  ...   14.57738  6.348691e+06
min     1.000000  2.503843e+06  ...    1.00000  2.750105e+06
25%    13.250000  4.708718e+06  ...   13.25000  3.300611e+06
50%    26.500000  7.626439e+06  ...   25.50000  6.716353e+06
75%    38.750000  1.282468e+07  ...   37.75000  1.195822e+07
max    59.000000  3.638426e+07  ...   50.00000  3.303479e+07

[8 rows x 6 columns]
==================================================
=           df.describe(include='all')           =
==================================================
                                                    Airport  ...  Percent change 2010-2011
count                                                    50  ...                 50.000000
unique                                                   50  ...                       NaN
top     Atlanta, GA (Hartsfield-Jackson Atlanta Intern...  ...                       NaN
freq                                                      1  ...                       NaN
mean                                                    NaN  ...                -23.758000
std                                                     NaN  ...                  2.435963
min                                                     NaN  ...                -32.200000
25%                                                     NaN  ...                -25.275000
50%                                                     NaN  ...                -23.650000
75%                                                     NaN  ...                -22.075000
max                                                     NaN  ...                -19.500000

[11 rows x 9 columns]
==================================================
=                   df.info()                    =
==================================================
<class 'pandas.core.frame.DataFrame'>
Index: 50 entries, ATL to IND
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Airport                   50 non-null     object
 1   2001 Rank                 50 non-null     int64
 2   2001 Total                50 non-null     int64
 3   2010 Rank                 50 non-null     int64
 4   2010 Total                50 non-null     int64
 5   2011 Rank                 50 non-null     int64
 6    Total                    50 non-null     int64
 7   Percent change 2001-2011  50 non-null     float64
 8   Percent change 2010-2011  50 non-null     float64
dtypes: float64(2), int64(6), object(1)
memory usage: 3.9+ KB
None
```

Basic Indexing

- Similar to normal Python or numpy
- Slices select rows

One of the real strengths of pandas is the ability to easily select desired rows and columns. This can be done with simple subscripting, like normal Python, or extended subscripting, similar to numpy. In addition, pandas has special methods and attributes for selecting data.

For selecting columns, use the column name as the subscript value. This selects the entire column. To select multiple columns, use a sequence (list, tuple, etc.) of column names.

For selecting rows, use slice notation. This may not map to similar tasks in normal python. That is, dataframe[x:y] selects rows x through y, but dataframe[x] selects column x.

Example

pandas_selecting.py

```
import pandas as pd
from printheader import print_header

columns = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  # column labels
index = ['a', 'b', 'c', 'd', 'e', 'f']  # row labels

values = [  # sample data
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = pd.DataFrame(values, index=index, columns=columns)  # create dataframe with data,
row labels, and column labels
print_header('DataFrame df')
print(df, '\n')

print_header("df['alpha']")
print(df['alpha'], '\n')  # select column 'alpha' -- single value selects column by name

print_header("df.beta")
print(df.beta, '\n')  # same, but alternate syntax (only works if column name is letters,
digits, and underscores)

print_header("df[['alpha','epsilon','beta']]")
print(df[['alpha', 'epsilon', 'beta']])  # select columns -- note index is an iterable
print()

print_header("df['b':'e']")
print(df['b':'e'], '\n')  # select rows 'b' through 'e' using slice of row labels

print_header("df['b':'b']")
print(df['b':'b'], '\n')  # select row 'b' only using slice of row labels (returns
dataframe)

print_header("df[['alpha','epsilon','beta']]['b':'e']")
print(df[['alpha', 'epsilon', 'beta']]['b':'e'])  # select columns AND slice rows
print()
```

pandas_selecting.py

```
==================================================
=                  DataFrame df                  =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

==================================================
=                  df['alpha']                   =
==================================================
a    100
b    200
c    300
d    400
e    500
f    600
Name: alpha, dtype: int64

==================================================
=                    df.beta                     =
==================================================
a    110
b    210
c    310
d    410
e    510
f    610
Name: beta, dtype: int64

==================================================
=         df[['alpha','epsilon','beta']]         =
==================================================
   alpha  epsilon  beta
a    100      140   110
b    200      240   210
c    300      340   310
d    400      440   410
e    500      540   510
f    600      640   610

==================================================
=                  df['b':'e']                  =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540

==================================================
=                  df['b':'b']                  =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240

==================================================
=    df[['alpha','epsilon','beta']]['b':'e']     =
==================================================
   alpha  epsilon  beta
b    200      240   210
c    300      340   310
d    400      440   410
e    500      540   510
```

Saner indexing

- `loc[row-spec,col-spec]` for names (strings or numbers)
- `.iloc[row-spec,col-spec]` for 0-based position (integers only)
- `.loc[]` row or column specs can be
  - single name
  - iterable of names
  - range (inclusive) of names
- `.iloc[]` row or column specs can be
  - single number
  - iterable of numbers
  - range (exclusive) of numbers
- `.at[]` single value

The `.loc` and `.iloc` indexers provide more extensive and consistent selecting of rows and columns for dataframes. They both work exactly the same way, but `.loc` uses only row and column *names*, and `.iloc` uses only *positions*.

Both indexers use the *getitem* operator `[]`, with the syntax `[row-specifier, column-specifier]`.

For `.loc[]`, the specifier can be either a single name, an iterable of names, or a range of names. The end of a range is inclusive.

For `.iloc[]`, the specifier can be either a single numeric index (0-based), iterable of indexes, or a range of indexes. The end of a range is exclusive.

To select all rows, or all columns, use `:`.

The `.at[]` property can be used to select a single value at a given row and column: `df.at[47, "color"]`.

NOTE The column specifier can be omitted, which will select all columns for those rows.

Example

pandas_loc.py

```python
import pandas as pd
from printheader import print_header


cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
indices = ['a', 'b', 'c', 'd', 'e', 'f']

values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = pd.DataFrame(values, index=indices, columns=cols)
print_header('DataFrame df')
print(df, '\n')

print_header("df.loc['b', 'delta']")  # one value
print(df.loc['b', 'delta'], "\n")


print_header("df.loc['b']")  # one row
print(df.loc['b'], '\n')

print_header("df.loc[:,'delta']")  # one column
print(df.loc[:,'delta'], '\n')


print_header("df.loc['b': 'd']")  # range of rows
print(df.loc['b':'d', :], '\n')
print(df.loc['b':'d'], '\n')   # shorter version

print_header("df.loc[:,'beta':'delta'")  # range of columns
print(df.loc[:, 'beta':'delta'], "\n")

print_header("df.loc['b':'d', 'beta':'delta']")  # ranges of rows and columns
print(df.loc['b':'d', 'beta':'delta'], '\n')

print_header("df.loc[['b', 'e', 'a']]")  # iterable of rows
print(df.loc[['b', 'e', 'a']], "\n")
```

```
print_header("df.loc[:, ['gamma', 'alpha', 'epsilon']]")  # iterable of columns
print(df.loc[:, ['gamma', 'alpha', 'epsilon']], "\n")

print_header("df.loc[['b', 'e', 'a'], ['gamma', 'alpha', 'epsilon']]")  # iterables of
rows and columns
print(df.loc[['b', 'e', 'a'], ['gamma', 'alpha', 'epsilon']], "\n")
```

pandas_loc.py

```
==================================================
=                  DataFrame df                  =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

==================================================
=              df.loc['b', 'delta']              =
==================================================
230

==================================================
=                  df.loc['b']                   =
==================================================
alpha      200
beta       210
gamma      220
delta      230
epsilon    240
Name: b, dtype: int64

==================================================
=               df.loc[:,'delta']                =
==================================================
a    130
b    230
c    330
d    430
e    530
f    630
Name: delta, dtype: int64
```

```
==================================================
=                df.loc['b': 'd']                =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440

   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440

==================================================
=            df.loc[:,'beta':'delta'             =
==================================================
   beta  gamma  delta
a   110    120    130
b   210    220    230
c   310    320    330
d   410    420    430
e   510    520    530
f   610    620    630

==================================================
=        df.loc['b':'d', 'beta':'delta']         =
==================================================
   beta  gamma  delta
b   210    220    230
c   310    320    330
d   410    420    430

==================================================
=            df.loc[['b', 'e', 'a']]             =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
e    500   510    520    530      540
a    100   110    120    130      140

==================================================
=    df.loc[:, ['gamma', 'alpha', 'epsilon']]    =
==================================================
   gamma  alpha  epsilon
a    120    100      140
b    220    200      240
c    320    300      340
```

```
d    420    400      440
e    520    500      540
f    620    600      640

==================================================
df.loc[['b', 'e', 'a'], ['gamma', 'alpha', 'epsilon']]
==================================================
   gamma  alpha  epsilon
b    220    200      240
e    520    500      540
a    120    100      140
```

Example

pandas_iloc.py

```
import pandas as pd
from printheader import print_header


cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
indices = ['a', 'b', 'c', 'd', 'e', 'f']

values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = pd.DataFrame(values, index=indices, columns=cols)
print_header('DataFrame df')
print(df, '\n')

print_header("df.iloc[1, 3]")  # one value
print(df.iloc[1, 3], "\n")


print_header("df.iloc[1]")  # one row
print(df.iloc[1], '\n')

print_header("df.iloc[:,3]")  # one column
print(df.iloc[:, 3], '\n')


print_header("df.iloc[1: 3]")  # range of rows
print(df.iloc[1:3, :], '\n')
print(df.iloc[1:3], '\n')   # shorter version

print_header("df.iloc[:,1:3]")  # range of columns
print(df.iloc[:, 1:3], "\n")

print_header("df.iloc[1:3, 1:3]")  # ranges of rows and columns
print(df.iloc[1:3, 1:3], '\n')

print_header("df.iloc[[1, 4, 0]]")  # iterable of rows
print(df.iloc[[1, 4, 0]], "\n")
```

```
print_header("df.iloc[:, [2, 0, 4]]")  # iterable of columns
print(df.iloc[:, [2, 0, 4]], "\n")

print_header("df.iloc[[1, 4, 0], [2, 0, 4]]")  # iterables of rows and columns
print(df.iloc[[1, 4, 0], [2, 0, 4]], "\n")
```

pandas_iloc.py

```
==================================================
=                  DataFrame df                  =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

==================================================
=                 df.iloc[1, 3]                  =
==================================================
230

==================================================
=                   df.iloc[1]                   =
==================================================
alpha      200
beta       210
gamma      220
delta      230
epsilon    240
Name: b, dtype: int64

==================================================
=                  df.iloc[:,3]                  =
==================================================
a    130
b    230
c    330
d    430
e    530
f    630
Name: delta, dtype: int64
```

```
==================================================
=                df.iloc[1: 3]                   =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340

   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340

==================================================
=                df.iloc[:,1:3]                  =
==================================================
   beta  gamma
a   110    120
b   210    220
c   310    320
d   410    420
e   510    520
f   610    620

==================================================
=              df.iloc[1:3, 1:3]                 =
==================================================
   beta  gamma
b   210    220
c   310    320

==================================================
=              df.iloc[[1, 4, 0]]                =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
e    500   510    520    530      540
a    100   110    120    130      140

==================================================
=            df.iloc[:, [2, 0, 4]]               =
==================================================
   gamma  alpha  epsilon
a    120    100      140
b    220    200      240
c    320    300      340
d    420    400      440
e    520    500      540
f    620    600      640
```

```
==================================================
=         df.iloc[[1, 4, 0], [2, 0, 4]]          =
==================================================
   gamma  alpha  epsilon
b    220    200      240
e    520    500      540
a    120    100      140
```

Broadcasting

- Operation is applied across rows and columns
- Can be restricted to selected rows/columns
- Sometimes called vectorization
- Use apply() for more complex operations

If you multiply a dataframe by some number, the operation is broadcast, or vectorized, across all values. This is true for all basic math operations.

The operation can be restricted to selected columns.

For more complex operations, the apply() method will apply a function that selects elements. You can use the name of an existing function, or supply a lambda (anonymous) function.

Example

pandas_broadcasting.py

```
import pandas as pd
from printheader import print_header

column_labels = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  # column labels
row_labels = pd.date_range('2013-01-01 00:00:00', periods=6, freq='D')  # date range to
be used as row indexes

print(row_labels, "\n")

values = [  # sample data
    [100, 110, 120, 930, 140],
    [250, 210, 120, 130, 840],
    [300, 310, 520, 430, 340],
    [275, 410, 420, 330, 777],
    [300, 510, 120, 730, 540],
    [150, 610, 320, 690, 640],
]

df = pd.DataFrame(values, row_labels, column_labels)  # create dataframe from data
print_header("Basic DataFrame:")
print(df)
print()

print_header("Triple each value")
print(df * 3)
print()  # multiply every value by 3

print_header("Multiply column gamma by 1.5")
df['gamma'] *= 1.5  # multiply values in column 'gamma' by 1.
print(df)
print()
```

pandas_broadcasting.py

```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

==================================================
=                Basic DataFrame:                =
==================================================
            alpha  beta  gamma  delta  epsilon
2013-01-01    100   110    120    930      140
2013-01-02    250   210    120    130      840
2013-01-03    300   310    520    430      340
2013-01-04    275   410    420    330      777
2013-01-05    300   510    120    730      540
2013-01-06    150   610    320    690      640

==================================================
=               Triple each value                =
==================================================
            alpha  beta  gamma  delta  epsilon
2013-01-01    300   330    360   2790      420
2013-01-02    750   630    360    390     2520
2013-01-03    900   930   1560   1290     1020
2013-01-04    825  1230   1260    990     2331
2013-01-05    900  1530    360   2190     1620
2013-01-06    450  1830    960   2070     1920

==================================================
=          Multiply column gamma by 1.5          =
==================================================
            alpha  beta  gamma  delta  epsilon
2013-01-01    100   110  180.0    930      140
2013-01-02    250   210  180.0    130      840
2013-01-03    300   310  780.0    430      340
2013-01-04    275   410  630.0    330      777
2013-01-05    300   510  180.0    730      540
2013-01-06    150   610  480.0    690      640
```

Counting unique occurrences

- Use `.value_counts()`
- Called from column

To count the unique occurrences within a column, call the method `value_counts()` on the column. It returns a `Series` object with the column values and their counts.

Example

pandas_unique.py

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_excel('https://qrc.depaul.edu/Excel_Files/Presidents.xlsx',
sheet_name='Master',
                    na_values='NA()')
df.index = range(1,len(df)+1)

print(df.head())
print(df.loc[1])
party_counts = df['Political Party'].value_counts()
print(party_counts)
# plot the data
plt.figure(figsize=(20.0,8.0))
party_counts.plot(kind='barh')
plt.show()
```

Creating new columns

- Assign to column with new name
- Use normal operators with other columns

For simple cases, it’s easy to create new columns. Just assign a Series-like object to a new column name. The easy way to do this is to combine other columns with an operator or function.

Example

pandas_new_columns.py

```python
import pandas as pd

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
index = ['a', 'b', 'c', 'd', 'e', 'f']

values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = pd.DataFrame(values, index=index, columns=cols)

def times_ten(x):
    return x * 10

df['zeta'] = df['delta'] * df['epsilon'] # product of two columns
df['eta'] = times_ten(df.alpha) # user-defined function
df['theta'] = df.sum(axis=1)  # sum each row
df['iota'] = df.mean(axis=1)  # avg of each row
df['kappa'] = df.loc[:,'alpha':'epsilon'].mean(axis=1)
# column kappa is avg of selected columns

print(df)
```

pandas_new_columns.py

```
   alpha  beta  gamma  delta  epsilon    zeta   eta   theta      iota  kappa
a    100   110    120    130      140   18200  1000   19800    4950.0  120.0
b    200   210    220    230      240   55200  2000   58300   14575.0  220.0
c    300   310    320    330      340  112200  3000  116800   29200.0  320.0
d    400   410    420    430      440  189200  4000  195300   48825.0  420.0
e    500   510    520    530      540  286200  5000  293800   73450.0  520.0
f    600   610    620    630      640  403200  6000  412300  103075.0  620.0
```

Removing entries

- Remove rows or columns
- Use drop() method

To remove columns or rows, use the `drop()` method, with the appropriate labels. Use `axis=1` to drop columns, or axis=0 to drop rows.

Example

pandas_drop.py

```
import pandas as pd
from printheader import print_header

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
index = ['a', 'b', 'c', 'd', 'e', 'f']
values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]
print_header('values:')
print(values, '\n\n')

df = pd.DataFrame(values, index=index, columns=cols)  # create dataframe
print_header('DataFrame df')
print(df, '\n')

df2 = df.drop(['beta', 'delta'], axis=1)  # drop columns beta and delta (axes: 0=rows,
1=columns)
print_header("After dropping beta and delta:")
print(df2, '\n')

print_header("After dropping rows b, c, and e")
df3 = df.drop(['b', 'c', 'e'])  # drop rows b, c, and e
print(df3)

print_header(" In-place drop")
df.drop(['beta', 'gamma'], axis=1, inplace=True)
print(df)
df.drop(['b', 'c'], inplace=True)
print(df)
```

pandas_drop.py

```
==================================================
=                    values:                     =
==================================================
[[100, 110, 120, 130, 140], [200, 210, 220, 230, 240], [300, 310, 320, 330, 340], [400,
410, 420, 430, 440], [500, 510, 520, 530, 540], [600, 610, 620, 630, 640]]



==================================================
=                  DataFrame df                  =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

==================================================
=          After dropping beta and delta:        =
==================================================
   alpha  gamma  epsilon
a    100    120      140
b    200    220      240
c    300    320      340
d    400    420      440
e    500    520      540
f    600    620      640

==================================================
=         After dropping rows b, c, and e        =
==================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
d    400   410    420    430      440
f    600   610    620    630      640
==================================================
=                  In-place drop                 =
==================================================
   alpha  delta  epsilon
a    100    130      140
b    200    230      240
c    300    330      340
d    400    430      440
e    500    530      540
```

```
f    600    630      640
   alpha  delta  epsilon
a    100    130      140
d    400    430      440
e    500    530      540
f    600    630      640
```

Useful pandas methods

Table 22. Methods and attributes for fetching DataFrame/Series data

Method	Description
`DF.columns()`	Get or set column labels

`DF.shape()`
`S.shape()`
	Get or set shape (length of each axis)

`DF.head(n)`
`DF.tail(n)`
	Return n items (default 5) from beginning or end

`DF.describe()`
`S.describe()`
	Display statistics for dataframe

`DF.info()`	Display column attributes
`DF.values` `S.values`	Get the actual values from a data structure
`DF.loc[row_indexer`[1]`, col_indexer]`	Multi-axis indexing by label (not by position)
`DF.iloc[row_indexer`[2]`, col_indexer]`	Multi-axis indexing by position (not by labels)

[1] Indexers can be label, slice of labels, or iterable of labels.
[2] Indexers can be numeric index (0-based), slice of indexes, or iterable of indexes.

Table 23. Methods for Computations/Descriptive Stats (called from pandas)

Method	Returns
`abs()`	absolute values
`corr()`	pairwise correlations
`count()`	number of values
`cov()`	Pairwise covariance
`cumsum()`	cumulative sums
`cumprod()`	cumulative products
`cummin(), cummax()`	cumulative minimum, maximum
`kurt()`	unbiased kurtosis
`median()`	median
`min(), max()`	minimum, maximum values
`prod()`	products
`quantile()`	values at given quantile
`skew()`	unbiased skewness
`std()`	standard deviation
`var()`	variance

NOTE these methods return Series or DataFrames, as appropriate, and can be computed over rows (axis=0) or columns (axis=1). They generally skip NA/null values.

Even more pandas…

At this point, please load the following Jupyter notebooks for more pandas exploration:

- pandas_Demo.ipynb
- pandas_Input_Demo.ipynb
- pandas_Selection_Demo.ipynb

NOTE The instructor will explain how to start the Jupyter server.

Chapter 16 Exercises

Exercise 16-1 (add_columns.py)

Read in the file **sales_records.csv** as shown in the early part of the chapter. Add three new columns to the dataframe:

- Total Revenue (*units sold x unit price*)
- Total Cost (*units sold x unit cost*)
- Total Profit (*total revenue - total cost*)

Exercise 16-2 (parasites.py))

The file parasite_data.csv, in the DATA folder, has some results from analysis on some intestinal parasites (not that it matters for this exercise…). Read parasite_data.csv into a DataFrame. Print out all rows where the Shannon Diversity is >= 1.0.

Chapter 17: Introduction to Matplotlib

Objectives

- Understand what matplotlib can do
- Create many kinds of plots
- Label axes, plots, and design callouts

About matplotlib

- matplotlib is a package for making 2D plots
- Emulates MATLAB®, but not a drop-in replacement
- matplotlib's philosophy: create simple plots simply
- Plots are publication quality
- Plots can be rendered in GUI applications

This chapter's discussion of matplotlib will use the iPython notebook named **MatplotlibExamples.ipynb**. Please start the iPython notebook server and load this notebook, as directed by the instructor.

matplotlib architecture

- pylab/pyplot front end plotting functions
- API create/manage figures, text, plots
- backends device-independent renderers

matplotlib consists of roughly three parts: pylab/pyplot, the API, and and the backends.

pyplot is a set of functions which allow the user to quickly create plots. Pyplot functions are named after similar functions in MATLAB.

The API is a large set of classes that do all the work of creating and manipulating plots, lines, text, figures, and other graphic elements. The API can be called directly for more complex requirements.

pylab combines pyplot with numpy. This makes pylab emulate MATLAB more closely, and thus is good for interactive use, e.g., with iPython. On the other hand, pyplot alone is very convenient for scripting. The main advantage of pylab is that it imports methods from both pyplot and pylab.

There are many backends which render the in-memory representation, created by the API, to a video display or hard-copy format. For example, backends include PS for Postscript, SVG for scalable vector graphics, and PDF.

The normal import is

```
import matplotlib.pyplot as plt
```

Matplotlib Terminology

- Figure
- Axis
- Subplot

A Figure is one "picture". It has a border ("frame"), and other attributes. A Figure can be saved to a file.

A Plot is one set of values graphed onto the Figure. A Figure can contain more than one Plot.

Axes and Subplot are similar; the difference is how they get placed on the figure. Subplots allow multiple plots to be placed in a rectangular grid. Axes allow multiple plots to placed at any location, including within other plots, or overlapping.

matplotlib uses default objects for all of these, which are sufficient for simple plots. You can explicitly create any or all of these objects to fine-tune a graph. Most of the time, for simple plots, you can accept the defaults and get great-looking figures.

Matplotlib Keeps State

- Primary method is matplotlib.pyplot()
- The current figure can have more than one plot
- Calling show() displays the current figure

matplotlib.pyplot is the workhorse of figure drawing. It is usually aliased to "plt".

While Matplotlib is object oriented, and you can manually create figures, axes, subplots, etc., pyplot() will create a figure object for you automatically, and commands called from pyplot() (usually through the **plt** alias) will work on that object.

Calling **plt.plot()** plots one set of data on the current figure. Calling it again adds another plot to the same figure.

plt.show() displays the figure, although iPython may display each separate plot, depending on the current settings.

You can pass one or two datasets to plot(). If there are two datasets, they need to be the same length, and represent the x and y data.

What Else Can You Do?

- Multiple plots
- Control ticks on any axis
- Scatter plots
- Polar axes
- 3D Plots
- Quiver plots
- Pie Charts

There are many other types of drawings that matplotlib can create. Also, there are many more style details that can be tweaked. See http://matplotlib.org/gallery.html for dozens of sample plots and their source.

There are many extensions (AKA toolkits) for Matplotlib, including Seaborne, CartoPy, at Natgrid.

Matplotlib Demo

At this point, please open the notebook **MatPlotLibExamples.ipynb** for an instructor-led tour of MPL features.

Chapter 17 Exercises

Exercise 17-1 (energy_use_plot.py)

Using the file energy_use_quad.csv in the DATA folder, use matplotlib to plot the data for "Transportation", "Industrial", and "Residential and Commercial". Don’t plot the "as a percent…".

You can do this in iPython, or as a standalone script. If you create a standalone script, save the figure to a file, so you can view it.

Use pandas to read the data. The columns are, in Python terms:

```
['Desc',"1960","1965","1970","1975","1980","1985","1990","1991","1992","1993","1994","199
5","1996","1997","1998","1999","2000","2001","2002","2003","2004","2005","2006","2007","2
008","2009","2010","2011"]
```

TIP See the script pandas_energy.py in the EXAMPLES folder to see how to load the data.

Appendix A: Python Bibliography

Data Science

- ***Building machine learning systems with Python***. William Richert, Luis Pedro Coelho. Packt Publishing
- ***High Performance Python***. Mischa Gorlelick and Ian Ozsvald. O'Reilly Media
- ***Introduction to Machine Learning with Python***. Sarah Guido. O'Reilly & Assoc.
- ***iPython Interactive Computing and Visualization Cookbook***. Cyril Rossant. Packt Publishing
- ***Learning iPython for Interactive Computing and Visualization***. Cyril Rossant. Packt Publishing
- ***Learning Pandas***. Michael Heydt. Packt Publishing
- ***Learning scikit-learn: Machine Learning in Python***. Raúl Garreta, Guillermo Moncecchi. Packt Publishing
- ***Mastering Machine Learning with Scikit-learn***. Gavin Hackeling. Packt Publishing
- ***Matplotlib for Python Developers***.Sandro Tosi.Packt Publishing
- ***Numpy Beginner's Guide.Ivan Idris***.Packt Publishing
- ***Numpy Cookbook.Ivan Idris***.Packt Publishing
- ***Practical Data Science Cookbook.Tony Ojeda, Sean Patrick Murphy, Benjamin Bengfort, Abhijit Dasgupta***.Packt Publishing
- ***Python Text Processing with NLTK 2.0 Cookbook.Jacob Perkins***.Packt Publishing
- ***Scikit-learn cookbook.Trent Hauck***.Packt Publishing
- ***Python Data Visualization Cookbook.Igor Milovanovic***.Packt Publishing
- ***Python for Data Analysis.Wes McKinney.***. O'Reilly & Assoc

Design Patterns

- ***Design Patterns: Elements of Reusable Object-Oriented Software.Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides***.Addison-Wesley Professional
- ***Head First Design Patterns.Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra***.O'Reilly Media
- ***Learning Python Design Patterns.Gennadiy Zlobin***.Packt Publishing
- ***Mastering Python Design Patterns.Sakis Kasampalis***.Packt Publishing

General Python development

- ***Expert Python Programming.Tarek Ziadé***.Packt Publishing
- ***Fluent Python.Luciano Ramalho***. O'Reilly & Assoc.

- ***Learning Python, 2nd Ed..Mark Lutz, David Asher***. O'Reilly & Assoc.
- ***Mastering Object-oriented Python.Stephen F. Lott***.Packt Publishing
- ***Programming Python, 2nd Ed. .Mark Lutz***. O'Reilly & Assoc.
- ***Python 3 Object Oriented Programming.Dusty Phillips***.Packt Publishing
- ***Python Cookbook, 3rd. Ed.. David Beazley, Brian K. Jones***. O'Reilly & Assoc.
- ***Python Essential Reference, 4th. Ed..David M. Beazley***.Addison-Wesley Professional
- ***Python in a Nutshell.Alex Martelli***. O'Reilly & Assoc.
- ***Python Programming on Win32.Mark Hammond, Andy Robinson***. O'Reilly & Assoc.
- ***The Python Standard Library By Example.Doug Hellmann***.Addison-Wesley Professional

Misc

- ***Python Geospatial Development.Erik Westra***.Packt Publishing
- ***Python High Performance Programming.Gabriele Lanaro***.Packt Publishing

Networking

- ***Python Network Programming Cookbook.Dr. M. O. Faruque Sarker***.Packt Publishing
- ***Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers.T J O'Connor***.Syngress
- ***Web Scraping with Python.Ryan Mitchell***.O'Reilly & Assoc.

Testing

- ***Python Testing Cookbook.Greg L. Turnquist***.Packt Publishing
- ***Learning Python Testing.Daniel Arbuckle***.Packt Publishing
- ***Learning Selenium Testing Tools, 3rd Ed. .Raghavendra Prasad MG***.Packt Publishing

Web Development

- ***Building Web Applications with Flask.Italo Maia***.Packt Publishing
- ***Django 1.0 Website Development.Ayman Hourieh***.Packt Publishing
- ***Django 1.1 Testing and Development.Karen M. Tracey***.Packt Publishing
- ***Django By Example.Antonio Melé***.Packt Publishing
- ***Django Design Patterns and Best Practices.Arun Ravindran***.Packt Publishing
- ***Django Essentials.Samuel Dauzon***.Packt Publishing

- ***Django Project Blueprints.Asad Jibran Ahmed***.Packt Publishing
- ***Flask Blueprints.Joel Perras***.Packt Publishing
- ***Flask by Example.Gareth Dwyer***.Packt Publishing
- ***Flask Framework Cookbook.Shalabh Aggarwal***.Packt Publishing
- ***Flask Web Development.Miguel Grinberg***. O'Reilly & Assoc.
- ***Full Stack Python (e-book only).Matt Makai***.Gumroad (or free download)
- ***Full Stack Python Guide to Deployments (e-book only).Matt Makai***.Gumroad (or free download)
- ***High Performance Django.Peter Baumgartner, Yann Malet***.Lincoln Loop
- ***Instant Flask Web Development.Ron DuPlain***.Packt Publishing
- ***Learning Flask Framework.Matt Copperwaite, Charles O Leifer***.Packt Publishing
- ***Mastering Flask.Jack Stouffer***.Packt Publishing
- ***Two Scoops of Django: Best Practices for Django 1.11. Daniel Roy Greenfeld, Audrey Roy Greenfeld***.Two Scoops Press
- ***Web Development with Django Cookbook.Aidas Bendoraitis***.Packt Publishing

Appendix B: Chapter 18: Network Programming

Objectives

- Download web pages or file from the Internet
- Consume web services
- Send e-mail using a mail server
- Learn why requests is the best HTTP client

Making HTTP requests

- Use the **requests** module
- Pythonic front end to urllib, urllib2, httplib, etc
- Makes HTTP transactions simple

The standard library provides the **urllib** package. It and its friends are powerful libraries, but their interfaces are complex for non-trivial tasks. There is a lot of code to write if you want to provide authentication, proxies, headers, or data, among other things.

The **requests** module is a much easier to use HTTP client module. It is included with the **Anaconda** distribution, or is readily available from **PyPI**.

requests implements GET, POST, PUT, and other HTTP verbs, and takes care of all the protocol housekeeping needed to send data on the URL, to send a username/password, and to retrieve data in various formats.

To use **requests**, import the module and then call **requests**.***VERB***, where *VERB* is "get", "post", "put", "patch", "delete", or "head". The first argument to any of these methods is the URL, followed by any of the named parameters for fine-tuning the request.

These methods return an HTTPResponse object, which contains the headers and data returned from the HTTP server. If the URL refers to a web page, then the **text** attribute contains the text of the page as a Python string.

In all cases, the **content** attribute contains the raw content from the server as a **bytes** string. If the returned data is a JSON string, the **json()** method converts the JSON data into a Python nested list or dictionary.

The **status_code** attribute contains the HTTP status code, normally 200 for a successful request.

For GET requests, URL parameters can be specified as a dictionary, using the **params** parameter.

For POST, PUT, or PATCH requests, the data to be uploaed can be specified as a dictionary using the **data** parameter.

TIP See details of the **requests** API at http://docs.python-requests.org/en/v3.0.0/api/#main-interface

Example

read_html_requests.py

```
import requests

URL = 'http://www.python.org'

response = requests.get(URL)

if response.status_code == requests.codes.OK:

    for header, value in sorted(response.headers.items()): # response.headers is a
dictionary of the headers
        print("{:20.20s} {}".format(header, value))
    print()

    print(response.text[:200])   # The text is returned as a bytes object, so it needs to
be decoded to a string; print the first 200 bytes
    print('...')
    print(response.text[-200:])   # print the last 200 bytes
```

Example

read_pdf_requests.py

```
import sys
import os
from subprocess import run  # for running external PDF viewer
import requests

url = 'https://www.nasa.gov/pdf/739318main_ISS%20Utilization%20Brochure
%202012%20Screenres%203-8-13.pdf'  # target URL
saved_pdf_file = 'nasa_iss.pdf'  # name of PDF file for saving

response = requests.get(url)  # open the URL
if response.status_code == requests.codes.OK:  # check status code
    if response.headers.get('content-type') == 'application/pdf':
        with open(saved_pdf_file, 'wb') as pdf_out:  # open local file
            pdf_out.write(response.content)  # write data to a local file in binary mode;
response.content is data from URL

        if sys.platform == 'win32':  # select platform and choose the app to open the PDF
file
            cmd = saved_pdf_file
        elif sys.platform == 'darwin':
            cmd = 'open ' + saved_pdf_file
        else:
            cmd = 'acroread ' + saved_pdf_file

        run(cmd, shell=True)  # run command with command.exe or Mac/Linux shell
```

Example

web_content_consumer_requests.py

```
import sys
import requests

BASE_URL = 'https://www.dictionaryapi.com/api/v3/references/collegiate/json/'  # base URL
of resource site

with open('dictionaryapikey.txt') as api_key_in:
    API_KEY = api_key_in.read().rstrip()  # get credentials


def main(args):
    if len(args) < 1:
        print("Please specify a search term")
        sys.exit(1)

    response = requests.get(
        BASE_URL + args[0],
        params={'key': API_KEY},
        # ssl, proxy, cookies, headers, etc.
    )  # send HTTP request and get HTTP response

    if response.status_code == requests.codes.OK:  # 200?
        data = response.json()  # convert JSON content to Python data structure
        for entry in data: # check for results
            if isinstance(entry, dict):
                meta = entry.get("meta")
                if meta:
                    part_of_speech = '({})'.format(entry.get('fl'))
                    word_id = meta.get("id")
                    print("{} {}".format(word_id.upper(), part_of_speech))
                if "shortdef" in entry:
                    print('\n'.join(entry['shortdef']))
                print()
            else:
                print(entry)

    else:
        print("Sorry, HTTP response", response.status_code)

if __name__ == '__main__':
    main(sys.argv[1:])
```

web_content_consumer_requests.py wombat

```
WOMBAT (noun)
any of several stocky burrowing Australian marsupials (genera Vombatus and Lasiorhinus of
the family Vombatidae) resembling small bears
```

Table 24. Keyword Parameters for **requests** *methods*

Option	Data Type	Description
`allow_redirects`	`bool`	set to True if PUT/POST/DELETE redirect following is allowed
`auth`	`tuple`	authentication pair (user/token,password/key)
`cert`	`str or tuple`	path to cert file or ('cert', 'key') tuple
`cookies`	`dict or CookieJar`	cookies to send with request
`data`	`dict`	parameters for a POST or PUT request
`files`	`dict`	files for multipart upload
`headers`	`dict`	HTTP headers
`json`	`str`	JSON data to send in request body
`params`	`dict`	parameters for a GET request
`proxies`	`dict`	map protocol to proxy URL
`stream`	`bool`	if False, immediately download content
`timeout`	`float or tuple`	timeout in seconds or (connect timeout, read timeout) tuple
`verify`	`bool`	if True, then verify SSL cert

NOTE These can be used with any of the HTTP request types, as appropriate.

Table 25. `requests.Response` methods and attributes

Method/attribute	Definition
`apparent_encoding`	Returns the apparent encoding
`close()`	Closes the connection to the server
`content`	Content of the response, in bytes
`cookies`	A CookieJar object with the cookies sent back from the server
`elapsed`	A timedelta object with the time elapsed from sending the request to the arrival of the response
`encoding`	The encoding used to decode r.text
`headers`	A dictionary of response headers
`history`	A list of response objects holding the history of request (url)
`is_permanent_redirect`	True if the response is the permanent redirected url, otherwise False
`is_redirect`	True if the response was redirected, otherwise False
`iter_content()`	Iterates over the response
`iter_lines()`	Iterates over the lines of the response
`json()`	A JSON object of the result (if the result was written in JSON format, if not it raises an error)
`links`	The header links
`next`	A PreparedRequest object for the next request in a redirection
`ok`	True if status_code is less than 400, otherwise False
`raise_for_status()`	If an error occur, this method a HTTPError object
`reason`	A text corresponding to the status code
`request`	The request object that requested this response
`status_code`	A number that indicates the status (200 is OK, 404 is Not Found)
`text`	The content of the response, in unicode
`url`	The URL of the response

Authentication with requests

- Options
  - Basic-Auth
  - Digest
  - Custom
- Use **auth** argument

requests makes it eaasy to provide basic authentication to a web site.

In the simplest case, create a `requests.auth.HTTPBasicAuth` object with the username and password, then pass that to requests with the `auth` argument. Since this is a common use case, you can also just pass a `(user, password)` tuple to the `auth` parameter.

For digest authentication, use `requests.auth.HTTPDigestAuth` with the username and password.

For custom authentication, you can create your own auth class by inheriting from `requests.auth.AuthBase`.

For OAuth 1, OAuth 2, and OpenID, install `requests-oauthlib`. This additional module provides auth objects that can be passed in with the `auth` parameter, as above.

See https://docs.python-requests.org/en/latest/user/authentication/ for more details.

Example

basic_auth_requests.py

```python
import requests
from requests.auth import HTTPBasicAuth, HTTPDigestAuth

# base URL for httpbin
BASE_URL = 'https://httpbin.org'

# formats for httpbin
BASIC_AUTH_FMT = "/basic-auth/{}/{}"
DIGEST_AUTH_FMT = "/digest-auth/{}/{}/{}"

USERNAME = "spam"
PASSWORD = "ham"
BAD_PASSWORD = "toast"

REPORT_FMT = "{:35s} {}"

def main():
    basic_auth()
    digest()

def basic_auth():
    auth = HTTPBasicAuth(USERNAME, PASSWORD)
    response = requests.get(
        BASE_URL + BASIC_AUTH_FMT.format(USERNAME, PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Basic auth good password", response))

    response = requests.get(
        BASE_URL + BASIC_AUTH_FMT.format(USERNAME, PASSWORD),
        auth=(USERNAME, PASSWORD),
    )
    print(REPORT_FMT.format("Basic auth good password (shortcut)", response))

    response = requests.get(
        BASE_URL + BASIC_AUTH_FMT.format(USERNAME, BAD_PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Basic auth bad password", response))

def digest():
    auth = HTTPDigestAuth(USERNAME, PASSWORD)
    response = requests.get(
        BASE_URL + DIGEST_AUTH_FMT.format('WOMBAT', USERNAME, PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Digest auth good password", response))

    auth = HTTPDigestAuth(USERNAME, BAD_PASSWORD)
    response = requests.get(
        BASE_URL + DIGEST_AUTH_FMT.format('WOMBAT', USERNAME, PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Digest auth bad password", response))


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

basic_auth_requests.py

```
Basic auth good password             <Response [200]>
Basic auth good password (shortcut) <Response [200]>
Basic auth bad password              <Response [401]>
Digest auth good password            <Response [200]>
Digest auth bad password             <Response [401]>
```

Grabbing a web page the hard way

- import urlopen() from urllib.request
- urlopen() similar to open()
- Read response
- Use info() for metadata

While **requests** simplifies creating an HTTP client, the standard library module **urllib.request** includes **urlopen()**. It returns a file-like object. You can iterate over lines of HTML, or read all of the contents with read().

The URL is opened in binary mode ; you can download any kind of file which a URL represents – PDF, MP3, JPG, and so forth – by using read().

NOTE When downloading HTML or other text, a bytes object is returned; use decode() to convert it to a string.

In general, the preferred approach is to install and use **requests**.

Example

read_html_urllib.py

```
import urllib.request

u = urllib.request.urlopen("https://www.python.org")

print(u.info())  # .info() returns a dictionary of HTTP headers
print()

print(u.read(500).decode())   # The text is returned as a bytes object, so it needs to be
decoded to a string
```

read_html_urllib.py

```
Connection: close
Content-Length: 50273
Server: nginx
Content-Type: text/html; charset=utf-8
X-Frame-Options: SAMEORIGIN
Via: 1.1 vegur, 1.1 varnish, 1.1 varnish
Accept-Ranges: bytes
Date: Wed, 10 May 2023 13:48:11 GMT
Age: 771
X-Served-By: cache-iad-kiad7000025-IAD, cache-pdk17828-PDK
X-Cache: HIT, HIT
X-Cache-Hits: 77, 2
X-Timer: S1683726492.827580,VS0,VE0
Vary: Cookie
Strict-Transport-Security: max-age=63072000; includeSubDomains; preload




<!doctype html>
<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">   <![endif]-->
<!--[if IE 7]>      <html class="no-js ie7 lt-ie8 lt-ie9">          <![endif]-->
<!--[if IE 8]>      <html class="no-js ie8 lt-ie9">                 <![endif]-->
<!--[if gt IE 8]><!--><html class="no-js" lang="en" dir="ltr">  <!--<![endif]-->

<head>
    <!-- Google tag (gtag.js) -->
    <script async src="https://www.googletagmanager.com/gtag/js?id=G-
TF35YF9CVH"></script>
    <script>
      window.d
```

Example

read_pdf_urllib.py

```
import sys
import os
from urllib.request import urlopen
from urllib.error import HTTPError

# url to download a PDF file of a NASA ISS brochure

url = 'https://www.nasa.gov/pdf/739318main_ISS%20Utilization%20Brochure
%202012%20Screenres%203-8-13.pdf'  # target URL

saved_pdf_file = 'nasa_iss.pdf'  # name of PDF file for saving

try:
    URL = urlopen(url)  # open the URL
except HTTPError as e:  # catch any HTTP errors
    print("Unable to open URL:", e)
    sys.exit(1)

pdf_contents = URL.read()  # read all data from URL in binary mode
URL.close()

with open(saved_pdf_file, 'wb') as pdf_in:
    pdf_in.write(pdf_contents)  # write data to a local file in binary mode

if sys.platform == 'win32':  # select platform and choose the app to open the PDF file
    cmd = saved_pdf_file
elif sys.platform == 'darwin':
    cmd = 'open ' + saved_pdf_file
else:
    cmd = 'acroread ' + saved_pdf_file

os.system(cmd)  # launch the app
```

Consuming Web services the hard way

- Use urllib.parse to URL encode the query.
- Use urllib.request.Request
- Specify data type in header
- Open URL with urlopen Read data and parse as needed

To consume Web services, use the urllib.request module from the standard library. Create a urllib.request.Request object, and specify the desired data type for the service to return.

If needed, add a headers parameter to the request. Its value should be a dictionary of HTTP header names and values.

For URL encoding the query, use urllib.parse.urlencode(). It takes either a dictionary or an iterable of key/value pairs, and returns a single string in the format "K1=V1&K2=V2&..." suitable for appending to a URL.

Pass the Request object to urlopen(), and it will return a file-like object which you can read by calling its read() method.

The data will be a bytes object, so to use it as a string, call decode() on the data. It can then be parsed as appropriate, depending on the content type.

NOTE the example program on the next page queries the Merriam-Webster dictionary API. It requires a word on the command line, which will be looked up in the online dictionary.

TIP List of public RESTful APIs: http://www.programmableweb.com/apis/directory/1?protocol=REST

Example

web_content_consumer_urllib.py

```
"""
Fetch a word definition from Merriam-Webster's API
"""
import sys
from urllib.request import Request, urlopen
import json
# from pprint import pprint

DATA_TYPE = 'application/json'

with open('dictionaryapikey.txt') as api_key_in:
    API_KEY = api_key_in.read().rstrip()  # get credentials

URL_TEMPLATE = 'https://www.dictionaryapi.com/api/v3/references/collegiate/json/{}?key=
{}'  # base URL of resource site

def main(args):
    if len(args) < 1:
        print("Please specify a word to look up")
        sys.exit(1)

    search_term = args[0].replace(' ', '+')

    url = URL_TEMPLATE.format(search_term, API_KEY)  # build search URL

    do_query(url)

def do_query(url):
    print("URL:", url)
    request = Request(url)
    response = urlopen(request)  # send HTTP request and get HTTP response
    raw_json_string = response.read().decode()  # read content from web site and decode()
from bytes to str
    data = json.loads(raw_json_string)  # convert JSON string to Python data structure
    # print("RAW DATA:")
    # pprint(data)
    for entry in data:  # iterate over each entry in results
        if isinstance(entry, dict):
            meta = entry.get("meta") # retrieve items from results (JSON convert to lists
and dicts)
            if meta:
                part_of_speech = '({})'.format(entry.get('fl'))
                word_id = meta.get("id")
                print("{} {}".format(word_id.upper(), part_of_speech))
            if "shortdef" in entry:
                print('\n'.join(entry['shortdef']))
            print()
        else:
            print(entry)
if __name__ == '__main__':
    main(sys.argv[1:])
```

web_content_consumer_urllib.py wombat

```
URL: https://www.dictionaryapi.com/api/v3/references/collegiate/json/wombat?key=b619b55d-
faa3-442b-a119-dd906adc79c8
WOMBAT (noun)
any of several stocky burrowing Australian marsupials (genera Vombatus and Lasiorhinus of
the family Vombatidae) resembling small bears
```

sending e-mail

- import smtplib module
- Create an SMTP object specifying server
- Call sendmail() method from SMTP object

You can send e-mail messages from Python using the smtplib module. All you really need is one smtplib object, and one method – sendmail().

Create the smtplib object, then call the sendmail() method with the sender, recipient(s), and the message body (including any headers).

The recipients list should be a list or tuple, or could be a plain string containing a single recipient.

Example

email_simple.py

```python
from getpass import getpass  # module for hiding password
import smtplib  # module for sending email
from email.message import EmailMessage  # module for creating message
from datetime import datetime

TIMESTAMP = datetime.now().ctime()  # get a time string for the current date/time

SENDER = 'jstrick@mindspring.com'
RECIPIENTS = ['jstrickler@gmail.com']
MESSAGE_SUBJECT = 'Python SMTP example'

MESSAGE_BODY = """
Hello at {}.

Testing email from Python
""".format(TIMESTAMP)

SMTP_USER = 'pythonclass'
SMTP_PASSWORD = getpass("Enter SMTP server password:")  # get password (not echoed to
screen)

smtp = smtplib.SMTP("smtp2go.com", 2525)  # connect to SMTP server
smtp.login(SMTP_USER, SMTP_PASSWORD)  # log into SMTP server

msg = EmailMessage()  # create empty email message
msg.set_content(MESSAGE_BODY)  # add the message body
msg['Subject'] = MESSAGE_SUBJECT  # add the message subject
msg['from'] = SENDER  # add the sender address
msg['to'] = RECIPIENTS  # add a list of recipients

try:
    smtp.send_message(msg)  # send the message
except smtplib.SMTPException as err:
    print("Unable to send mail:", err)
finally:
    smtp.quit()  # disconnect from SMTP server
```

Email attachments

- Create MIME multipart message
- Create MIME objects
- Attach MIME objects
- Serialize message and send

To send attachments, you need to create a MIME multipart message, then create MIME objects for each of the attachments, and attach them to the main message. This is done with various classes provided by the **email.mime** module.

These modules include **multipart** for the main message, **text** for text attachments, **image** for image attachments, **audio** for audio files, and **application** for miscellaneous binary data.

One the attachments are created and attached, the message must be serialized with the **as_string()** method. The actual transport uses **smptlib**, just like simple email messages described earlier.

Example

email_attach.py

```python
import smtplib
from datetime import datetime
from imghdr import what  # module to determine image type
from email.message import EmailMessage # module for creating email message
from getpass import getpass # module for reading password privately


SMTP_SERVER = "smtp2go.com"  # global variables for external information (IRL should be
from environment -- command line, config file, etc.)
SMTP_PORT = 2525

SMTP_USER = 'pythonclass'

SENDER = 'jstrick@mindspring.com'
RECIPIENTS = ['jstrickler@gmail.com']


def main():
    smtp_server = create_smtp_server()
    now = datetime.now()
    msg = create_message(
        SENDER,
        RECIPIENTS,
        'Here is your attachment',
        'Testing email attachments from python class at {}\n\n'.format(now),
    )
    add_text_attachment('../DATA/parrot.txt', msg)
    add_image_attachment('../DATA/felix_auto.jpeg', msg)
    send_message(smtp_server, msg)


def create_message(sender, recipients, subject, body):
    msg = EmailMessage()  # create instance of EmailMessage to hold message
    msg.set_content(body)  # set content (message text) and various headers
    msg['From'] = sender
    msg['To'] = recipients
    msg['Subject'] = subject
    return msg


def add_text_attachment(file_name, message):
    with open(file_name) as file_in:  # read data for text attachment
        attachment_data = file_in.read()
    message.add_attachment(attachment_data)  # add text attachment to message


def add_image_attachment(file_name, message):
    with open(file_name, 'rb') as file_in:  # read data for binary attachment
        attachment_data = file_in.read()
    image_type = what(None, h=attachment_data)  # get type of binary data
    message.add_attachment(attachment_data, maintype='image', subtype=image_type)  # add
binary attachment to message, including type and subtype (e.g., "image/jpg)"


def create_smtp_server():
    password = getpass("Enter SMTP server password:")  # get password from user (don't
hardcode sensitive data in script)
    smtpserver = smtplib.SMTP(SMTP_SERVER, SMTP_PORT)  # create SMTP server connection
    smtpserver.login(SMTP_USER, password)  # log into SMTP connection

    return smtpserver


def send_message(server, message):
    try:
        server.send_message(message)  # send message
    finally:
        server.quit()


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

Example

email_html.py

```python
from getpass import getpass  # module for hiding password
import smtplib  # module for sending email
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

from datetime import datetime

TIMESTAMP = datetime.now().ctime()  # get a time string for the current date/time

SENDER = 'jstrick@mindspring.com'
RECIPIENTS = 'jstrickler@gmail.com,crgnc3@gmail.com'
MESSAGE_SUBJECT = 'Python SMTP example'

PLAIN_BODY = """
Hello at {}.

Testing email from Python
""".format(TIMESTAMP)

HTML_BODY = """
<html>
<head></head>
<body>
<h1>Hello</h1>
<h2>{}</h2>
<h3>Testing email from Python</h3>
<body>
</html>
""".format(TIMESTAMP)

plain_contents = MIMEText(PLAIN_BODY, "text")
html_contents = MIMEText(HTML_BODY, "html")


SMTP_USER = 'pythonclass'
SMTP_PASSWORD = getpass("Enter SMTP server password:")  # get password (not echoed to
screen)

smtp = smtplib.SMTP("smtp2go.com", 2525)  # connect to SMTP server
smtp.login(SMTP_USER, SMTP_PASSWORD)  # log into SMTP server

msg = MIMEMultipart('alternative')  # create empty email message
msg['Subject'] = MESSAGE_SUBJECT  # add the message subject
```

```
msg['from'] = SENDER  # add the sender address
msg['to'] = RECIPIENTS  # add a list of recipients

msg.attach(plain_contents)
msg.attach(html_contents)

try:
    smtp.sendmail(SENDER, RECIPIENTS, msg.as_string())  # send the message
except smtplib.SMTPException as err:
    print("Unable to send mail:", err)
finally:
    smtp.quit()  # disconnect from SMTP server
```

Remote Access

- Use paramiko (not part of standard library)
- Create ssh client
- Create transport object to use sftp and other tools

For remote access to other computers, you generally use the SSH protocol. Python has several ways to use SSH.

The current best way is to use paramiko. It is a pure-Python module for connecting to other computers using SSH. It is not part of the standard library, but is included with the Anaconda distribution.

NOTE

Paramiko is used by Ansible and other sys admin tools.

Find out more about paramiko at http://www.lag.net/paramiko/
Find out more about Ansible at http://www.ansible.com/
Find out more about **ssh2-python**, an alternative to Paramiko, at https://parallel-ssh.org/post/ssh2-python/

Auto-adding hosts

- Interactive SSH prompts to add new host
- Programmatic interface can't do that
- Use **set_missing_host_key_policy()**
- Adds to list of known hosts.

The first time you connect to a new host with SSH, you get the following message:

```
The authenticity of host HOSTNAME can't be established.
ECDSA key fingerprint is HOSTNAME
Are you sure you want to continue connecting...
```

To avoid the message when using Paramiko, call **set_missing_host_key_policy()** from the Paramiko SSH client object:

```
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
```

Remote commands

- Use SSHClient
- Access standard I/O channels

To run commands on a remote computer, use SSHClient. Once you connect to the remote host, you can execute commands and access the standard I/O of the remote program.

The **exec_command()** method executes a command on the remote host, and returns a 3-tuple with the remote command's stdin, stdout, and stderr as file-like objects.

You can read from stdout and stderr, and write to stdin.

NOTE With some versions of **paramiko**, the *stdin* object returned by **exec_command()** must be explicitly set to **None**, or deleted with **DEL** after use. Otherwise, an error will be raised.

Example

paramiko_commands.py

```
import paramiko

with paramiko.SSHClient() as ssh:  # create paramiko client

    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())  # ignore missing keys
(this is safe)

    ssh.connect('localhost', username='python', password='l0lz')  # connect to remote
host

    stdin, stdout, stderr = ssh.exec_command('whoami')  # execute remote command; returns
standard I/O objects
    print(stdout.read().decode())  # read stdout of command
    print('-' * 60)

    stdin, stdout, stderr = ssh.exec_command('ls -l')  # execute remote command; returns
standard I/O objects
    print(stdout.read().decode())  # read stdout of command
    print('-' * 60)

    stdin, stdout, stderr = ssh.exec_command('ls -l /etc/passwd /etc/horcrux')  # execute
remote command; returns standard I/O objects
    print("STDOUT:")
    print(stdout.read().decode())  # read stdout of command
    print("STDERR:")
    print(stderr.read().decode())  # read stderr of command
    print('-' * 60)

    del stdin  # workaround for paramiko bug!
```

paramiko_commands.py

```
python

------------------------------------------------------------
total 384
drwx------+  3 python  staff      96 Feb 11  2021 Desktop
drwx------+  3 python  staff      96 Feb 11  2021 Documents
drwx------+  3 python  staff      96 Feb 11  2021 Downloads
drwx------@ 52 python  staff    1664 Mar 29  2022 Library
drwx------+  3 python  staff      96 Feb 11  2021 Movies
drwx------+  3 python  staff      96 Feb 11  2021 Music
drwx------+  3 python  staff      96 Feb 11  2021 Pictures
drwxr-xr-x+  4 python  staff     128 Feb 11  2021 Public
-rw-r--r--   1 python  staff  148544 May 10 09:40 alice.txt
drwxr-xr-x   3 python  staff      96 May 10 09:40 text_files

------------------------------------------------------------
STDOUT:
-rw-r--r--  1 root  wheel  7868 Dec  2 03:43 /etc/passwd

STDERR:
ls: /etc/horcrux: No such file or directory

------------------------------------------------------------
```

Copying files with SFTP

- Create transport
- Create SFTP client with transport

To copy files with paramiko, first create a **Transport** object. Using a **with** block will automatically close the Transport object.

From the transport object you can create an SFTPClient. Once you have this, call standard FTP/SFTP methods on that object.

Some common methods include listdir_iter(), get(), put(), mkdir(), and rmdir().

Example

paramiko_copy_files.py

```
import os
import paramiko
from paramiko import Transport, SFTPClient

# In real life, don't hard-code password in script -- do one of these:
# with open('my_secret_file.txt') as secret_in:
#     password = secret_in.read().restrip()
# password = os.getenv("REMOTE_PASSWORD")

REMOTE_DIR = 'text_files'

with Transport(('localhost', 22)) as transport:  # create paramiko Transport instance
    transport.connect(username='python', password='l0lz')  # connect to remote host
    sftp = SFTPClient.from_transport(transport)  # create SFTP client using Transport
instance
    for item in sftp.listdir_iter():  # get list of items on default (login) folder
(listdir_iter() returns a generator)
        print(item)
    print('-' * 60)

    remote_files = sftp.listdir(REMOTE_DIR)  # delete remote dir if it exists
    print(f"remote_files: {remote_files}")
    if remote_files:
        for remote_file in remote_files:
            remote_path = os.path.join(REMOTE_DIR, remote_file)
            print(f"remote_path: {remote_path}")

            sftp.remove(remote_path)
        sftp.rmdir(REMOTE_DIR)

    # create remote dir
    sftp.mkdir(REMOTE_DIR)

    # sftp.put(local-file)
    # sftp.put(local-file, remote-file)
    remote_path = os.path.join(REMOTE_DIR, 'betsy.txt')  # create path for remote file
    sftp.put('../DATA/alice.txt', remote_path)  # create a folder on the remote host
    sftp.put('../DATA/alice.txt', 'alice.txt')
    sftp.get(remote_path, 'eileen.txt')  # copy a file to the remote host
    print(sftp.listdir(), '\n')
    print(sftp.listdir(REMOTE_DIR))
```

paramiko_copy_files.py

```
drwx------   1 503      20             96 11 Feb 2021  Music
-r--------   1 503      20              7 14 Sep 2021  .CFUserTextEncoding
drwx------   1 503      20             96 11 Feb 2021  Pictures
drwxr-xr-x   1 503      20             96 10 May 09:40 text_files
-rw-r--r--   1 503      20         148544 10 May 09:40 alice.txt
-rw-------   1 503      20            662 05 May 14:23 .zsh_history
drwx------   1 503      20             96 11 Feb 2021  Desktop
drwx------   1 503      20           1664 29 Mar 2022  Library
drwxr-xr-x   1 503      20            128 11 Feb 2021  Public
drwx------   1 503      20             96 11 Feb 2021  Movies
drwx------   1 503      20             96 11 Feb 2021  Documents
drwx------   1 503      20             96 11 Feb 2021  Downloads
------------------------------------------------------------
remote_files: ['betsy.txt']
remote_path: text_files/betsy.txt
['Music', '.CFUserTextEncoding', 'Pictures', 'text_files', 'alice.txt', '.zsh_history',
'Desktop', 'Library', 'Public', 'Movies', 'Documents', 'Downloads']

['betsy.txt']
```

Interactive remote access

- Write to stdin
- Read response from stdout

To interact with a remote program, write to the stdin object returned by ***ssh_object*.exec_command()**.

```
stdin.write("command input....\n")
```

Be sure to add a newline ('\n') for each line of input you send.

To get the response, read the next line(s) of code with *stdout*.readline()

Example

paramiko_interactive.py

```
import paramiko
# bc is an interactive calculator that comes with Unix-like systems (Linux, Mac, etc.)

with paramiko.SSHClient() as ssh:  # create paramiko SSH client
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())  # auto-add remote host

    ssh.connect('localhost', username='python', password='l0lz')  # log into to remote
host

    stdin, stdout, stderr = ssh.exec_command('bc')  # execute command; returns file-like
objects representing stdio

    stdin.write("17 + 25\n")  # write to command's stdin
    result = stdout.readline()  # read output of command
    print("Result is:", result)

    stdin.write("scale = 3\n")  # set scale (# decimal points) to 3 (bc-specific command)
    stdin.write("738.3/191.9\n")
    result = stdout.readline()
    print("Result is:", result)

    stdin.write("quit\n")  # create paramiko SSH client
    stdin = None   # auto-add remote host
```

paramiko_interactive.py

```
Result is: 42

Result is: 3.847
```

Chapter 18 Exercises

Exercise 18-1 (fetch_xkcd_requests.py, fetch_xkcd_urllib.py)

Write a script to fetch the following image from the Internet and display it. http://imgs.xkcd.com/comics/python.png

Exercise 18-2 (wiki_links_requests.py, wiki_links_urllib.py)

Write a script to count how many links are on the home page of Wikipedia. To do this, read the page into memory, then look for occurrences of the string "href".

TIP Use `string.count()` on the text of the web page.

NOTE For detailed screen-scraping, you can use the Beautiful Soup module.

Exercise 18-3 (send_chimp.py)

If the class conditions allow it (i.e., if you have access to the Internet, and an SMTP account), send an email to yourself with the image **chimp.bmp** (from the DATA folder) attached.

Appendix C: Git Access

What is source code control?

- Tracks changes to files
- Allows collaboration
- Provides for tagging a repo at given point

A source code control system tracks changes to files. When a file is first created, and then at any point afterwards, the file is *committed* to the code repository. Any previously committed version can be retrieved.

At a particular point in development, when all of the files are ready for production (or beta testing), you can *tag* the current state of the repository as "version X.y", so you can retrieve all files that comprise a particular version of the software.

Such systems are also designed for multiple developers to collaborate on a set of source files.

Other source code control systems include Subversion, Mercurial, and Microsoft Team Foundation Server.

What is git?

- Distributed source code control
- Keeps track of changes to your code
- Created by Linux Torvalds

git, created by Linux Torvalds, is the most popular modern source code control system.

git is distinguished from other systems by having no official "main" repository. It is designed for easy collaboration, and to make it easy to merge changes from files being updated by multiple developers.

With git, every developer has their own repo, and commits locally. Developers can also update to or from other repos.

git workflow

The normal git workflow can be divided into several areas.

Getting started

To get started, create a repo with `git init`. Then add any existing files with `git add …` to stage the files (tell git to track them) and `git commit …` (tell git to add to or update the repo).

Then as you create new files or make changes to files, commit the changes by repeating `git add …` and `git commit …`. You can update another repo with your changes with `git push`, or update your repo with someone else's with `git pull`.

Branching and merging

At some point, you'll want to create a *branch*. A branch is a separate copy of the repository. After working on the branch (e.g. for a bug fix or a new feature), you can *merge* your changes back into the primary repo.

Tagging

At any time, you can *tag* the repo with a label (e.g. "1.0") across all files, so that you can retrieve your entire project with a known state.

Using the command line

- git *command*
- Dozens of commands

A common way to use git is from the command line. git has many commands, each of which has options, arguments or subcommands.

For instance, to create a new git repository and add existing files:

```
git init
git add .
git commit -a -m "initial commit"
```

Later, you would typically repeat the following

```
git add foobar.py
git commit -m "added foobar.py" foobar.py
```

or

```
git add .
git commit -a -m "fixed issue #1234"
```

Common git commands: **add**, **commit**, **push**, **status**, **merge**, **branch**, **pull**, **fetch**, **clone**

TIP Many IDEs, such as PyCharm, Visual Studio Code, and Eclipse, have git integration built in.

NOTE see https://git-scm.com/doc for the full git documentation

Using the *gitpython* module

- Install **gitpython**
- Create Repo object
  - from path to repo
  - init new repo
- Create Git object from repo
- Methods map to git CLI commands

The **gitpython** module is a convenient wrapper for the git command line utility. To install with **pip**:

```
pip install gitpython
```

To get started, import the **Git** and **Repo** objects from **git**.

For an existing repo, initialize Repo with the path the repo. For a new repo, call Repo.init() with the desired path.

```
repo = Repo('/path/to/repo')
```

You can get some information, such as untracked files, from the Repo object, but for most tasks you'll need a Git object, which is initialized from the Repo object.

The Git object has methods that map to the commands of the **git** utility.

```
git = Git(repo)
print(git.status())
```

Example

git_status.py

```
import os
from git import Git, Repo

home_dir = os.path.expanduser('~')
repo_dir = os.path.join(home_dir, 'myproject')  # Folder of repository

repo = Repo(repo_dir)   # Create Repo object

os.chdir(repo_dir)  # Go to repo folder (not always required)

g = Git(repo)   # Create Git() object; this is used for most git commands
print(g.status())  # Get status
```

git_status.py

```
```

Getting repo info

- Use Repo object
- Many methods and properties
- Use *git*.log() for file changes

To get general information about a repo, you can access and instance of git.Repo.

Some of the information available is * Untracked files * List of branches * Whether the repo is "dirty" (has modified files) * List of remote repos

Use *git*.log() to print the log of recent changes to the codebase.

Example

git_info.py

```
import os
from git import Repo

home_dir = os.path.expanduser('~')
repo_dir = os.path.join(home_dir, 'myproject')

repo = Repo(repo_dir)

print("Untracked files:", repo.untracked_files, '\n')  # Get list of untracked files

print("Branches:")
for branch in repo.branches:  # Iterate over branch objects
    print(branch.name, branch.path)
print()

print(repo.is_dirty(), '\n')  # Are there any changed files?

print(repo.remotes)  # Get list of remotes
```

git_info.py

```
```

Example

git_log.py

```
import os
from git import Git, Repo

home_dir = os.path.expanduser('~')
repo_dir = os.path.join(home_dir, 'myproject')

repo = Repo(repo_dir)

os.chdir(repo_dir)

g = Git(repo)
print(g.log())  # get log for entire repo
print('-' * 60)
print(g.log('unicode.py'))  # get log for specific file
print('-' * 60)
print(g.log('fizzbuzz1.py'))  # get log for specific file
```

git_log.py

```
```

Creating a repo

- Use *repo*.init()
- Stage files with *git*.add()
- Commit files with *git*.commit()

Use *repo*.init() to create a new repo.

Use *git*.add(*filename*) to stage a particular file for commit, or *git*.add('.') to stage all untracked files in the repo.

Use *git*.commit(*filename*, "*commit message*") to commit one file, or *git*.commit('.', "*commit message*") to commit changes to all files that have been staged.

TIP Create a .gitignore file in the repo containing files and folders that git should not track.

TIP Use os.chdir() to change directory to the repo to avoid accidentally using a repo in a parent folder.

Example

git_complete.py

```python
import os
import shutil
from git import Git, Repo

home_dir = os.path.expanduser('~')
repo_dir = os.path.join(home_dir, 'myproject')

r = Repo.init(repo_dir)  # Create new empty repo
os.chdir(repo_dir)

file_to_add = 'creating_dicts.py'
g = Git(r)  # Create Git() object from repo
shutil.copy('../' + file_to_add, ".")  # Copy file to add (typically created with IDE, so
no need to copy)

g.add(file_to_add)  # Stage file for commit
g.commit(file_to_add, message="initial commit")  # Commit file
print(g.log())  # Show repo log
```

Pushing to remote repos

- Use_repo_.add_remote()

An important feature of **git** is the ability to update another repo from your local repo. Even though git is decentralized (i.e., does not depend on a central server), most organizations designate a "primary" repo that has the most up-to-date versions of a codebase.

The non-local repo is called, appropriately enough, the *remote*.

The remote repo must exist.

To add a remote, use *git*.remote('add', *remote_name*, *remote*), where *name* is the name and *remote* is the path (if the remote is on the same computer) or URL (if on a different machine) of the remote repo.

```
g.remote('add', 'origin', 'https://github.com/jstrickler/myproject.git')  ⑤
```

To automatically track the remote (i.e., push will default to the remote

```
g.push('-u', 'origin', 'master')
```

Now, *git*.push() with no arguments will update the remote.

You can have multiple remotes. In this case pass the name of the remote to *git*.push().

```
git.push()  # push local commits to configured remote
git.push("myremote") # push to specific remote
```

Example

git_remote.py

```
import os
from git import Git, Repo

home_dir = os.path.expanduser('~')
repo_dir = os.path.join(home_dir, 'myproject')

repo = Repo(repo_dir)

os.chdir(repo_dir)

g = Git(repo)

g.remote('add', 'origin', 'https://github.com/jstrickler/myproject.git')  # Add remote
repo named 'origin'

g.push('-u', 'origin', 'master')  # Push local files to remote and track 'origin' as the
"master" repo so push() will automatically use it
```

Cloning repos

- Use Repo.clone_from()

To clone (make a local copy of) a repo, use Repo.clone_from() with the URL of the repo you want to clone and the local folder to clone into.

```
Repo.clone_from("remote-URL", "local-path")
```

Example

git_clone.py

```
import os
from git import Git, Repo
#TODO: FIX!!!

github_url = 'https://github.com/jstrickler/myproject.git'
home_dir = os.path.expanduser('~')
repo_dir = os.path.join(home_dir, 'myproject')
clone_dir = os.path.join(home_dir, 'myproject_clone')

repo = Repo(repo_dir)

os.chdir(repo_dir)

Repo.clone_from(github_url, clone_dir)  # Clone existing repo to new repo
```

Working with branches

- Create branch for a particular task
- Merge branch back into main stream
- Use *git*.checkout()

Branches are used to make a "private" copy of a repo. git tracks changes to each branch separately, so updates to a branch don't affect other branches. This is so developers can collaborate on a codebase without changing each other's files.

To create a new branch and begin working on it:

```
git.checkout('-b', 'branch-name')
```

The name can be anything you want.

To work on an existing branch:

```
git.checkout('branch-name')
```

NOTE See https://nvie.com/posts/a-successful-git-branching-model/ for a good description of **git** workflow with branches.

Example

git_branch.py

```
import os
from git import Git, Repo

home_dir = os.path.expanduser('~')
repo_dir = os.path.join(home_dir, 'myproject')

repo = Repo(repo_dir)

os.chdir(repo_dir)

g = Git(repo)

g.checkout('-b', 'myfeature')  # Create new branch named "myfeature"

g.add('tyger.py')  # tyger.py is only added and committed to branch "myfeature", not main branch
g.commit('tyger.py', message='new file')
```

Tagging

- Creates a snapshot of the repo
- Mark all files in repo
- Creates specific version of codebase
- Use *repo*.create_tag()

At some point you'll want to "tag" the current state of a repository. This typically is tied to a "release" of a particular version of your project. A typical tag might be "v1.2.7a", representing major, minor, and micro versions of the project. However, the tag can be anything you want, such as "Beta9" or "weaseldust".

Tags are created from the Repo object, with create_tag().

```
repo.create_tag('v1.2.7a')
```

Once you have created a tag, you can then fetch, pull, or clone the repo and specify the tag. This will use the versions of the files at the time the tag was added.

Example

git_create_tag.py

```
import os
from git import Repo

home_dir = os.path.expanduser('~')
repo_dir = os.path.join(home_dir, 'myproject')

repo = Repo(repo_dir)

repo.create_tag("v1.0.0", message="First Public Release") # Create tag with descriptive
message
```

Example

git_list_tags.py

```
import os
from git import Repo

home_dir = os.path.expanduser('~')
repo_dir = os.path.join(home_dir, 'myproject')

repo = Repo(repo_dir)

for tag in repo.tags:  # Iterate over all tags for repo
    print(tag.name)
```

git_list_tags.py

```
```

Chapter 18 Exercises

Exercise 18-1 (funwithgit.py)

Clone https://github.com/jstrickler/myproject.git into a folder named myclone. Using PyCharm (or other IDE), create a new file in that folder named "your_name.py" (this is so everyone doesn’t use the same name). Add the new file, commit the file, and finally push the file to the remote.

Appendix D: Chapter 19: Design Patterns

Objectives

- Examine important OOP principles
- Understand why design patterns are useful
- Learn common design patterns
- Implement design patterns in Python

Coupling and cohesion

- Coupling: interdependence of two components
- Cohesion: how well components fit together
- Strive for looser coupling but tighter cohesion

Coupling and **cohesion** are two terms often used when studying design patterns. They refer to the way in which components interact.

Coupling describes how much (or how little) two components depend on each other. The more dependence, the greater the complexity, and the more that one has to change when the other changes. Components are tightly coupled if they share variables, or exchange information that alters their behavior.

Thus, one primary design goal for software components is looser coupling. This means that components can interact, but are independent of each other. An example of this is illustrated by the Strategy pattern.

Cohesion describes how well components fit together. In the ideal world, each component should implement just one function or provide just one data value. The more responsibilities a component has, the less cohesive it is.

Interfaces

- Do not exist as such in Python
- Also known as abstract classes (but not in Java)
- Can be implemented with the abc module

An important software design rule is to program to the **interface**, not the object.

Code which expects a particular object type is tightly coupled to that type, and any changes to the object type will require changes to the code.

For example, rather than building a logger that writes to a text file, it is better to build a logger that writes to an object that acts like, but is not necessarily, a text file. In some languages, we would say that the logger uses the text file interface, but Python does not directly support interfaces.

Being a dynamic language, Python uses "duck" typing, which means that any object providing the appropriate methods can be substituted for the expected object.

Thus, our logger can write to any object that implements a text-file-like interface, in the informal sense.

This is the Pythonic way, and works well, especially if components are well documented.

If you want to ensure that a class implements a particular set of methods, you can use the **abc** module. It allows you to create abstract base classes. These classes may not be directly instantiated, and if abstract methods are not overwritten, an exception is raised upon instantiation.

Whenever you want to enforce that "interface", add the abstract class to the base class list. This is possible because Python supports multiple inheritance.

Example

abstract_base_classes.py

```
from abc import ABCMeta, abstractmethod

class Animal(metaclass=ABCMeta):  # metaclasses control how classes are created; ABCMeta
adds restrictions to classes that inherit from Animal

    @abstractmethod   # when decorated with @abstractmethod, speak() becomes an abstract
method
    def speak(self):
        pass

class Dog(Animal):  # Inherit from abstract base class Animal
    def speak(self):   # speak() must be implemented
        print("woof! woof!")

class Cat(Animal):  # Inherit from abstract base class Animal
    def speak(self):  # speak() must be implemented
        print("Meow meow meow")

class Duck(Animal): # Inherit from abstract base class Animal
    pass  # Duck does not implement speak()

d = Dog()
d.speak()

c = Cat()
c.speak()

try:
    d = Duck()  # Duck throws a TypeError if instantiated
    d.speak()
except TypeError as err:
    print(err)
```

abstract_base_classes.py

```
woof! woof!
Meow meow meow
Can't instantiate abstract class Duck with abstract method speak
```

What are design patterns?

- Reusable solution pattern
- Not a design or specification
- Not an implementation
- A description of how to solve a problem
- A formalized best practice

A **design pattern** is basically a "way of doing something". Design patterns grow from observing the best ways to solve problems.

A design pattern is not a specification for a particular software node; it is a description of how to do something. It is more than an algorithm, because algorithms only describe logic. Design patterns describe the big-picture details of how to implement software for common use cases.

Design patterns became well-known in the computer science world after the publication of the book Design Patterns: Elements of Reusable Object-Oriented Software, published in 1994. The four authors are typically referred to as the "Gang of Four", and the book itself as the "GoF".

Design patterns are associated with object-oriented software, but are not restricted to OOP. The GoF book is pretty dry reading, and the examples are in Smalltalk and C++. For a newer take on design patterns, the following books are excellent:

- Mastering Python Design patterns
- Learning Python Design Patterns
- Head First Design Patterns (book examples are in Java)

NOTE find details for these books in Appendix A – Bibliography

Why use design patterns?

- Tried and true
- Vocabulary for discussing solutions
- Provide structure
- Higher-level than packages of actual code

Design patterns provide workable approaches to common coding problems. They help developers avoid mistakes, because the patterns have been discovered and refined by the actual experience of other programmers.

Another advantage is that design patterns provide a vocabulary for discussing the architecture of a group of related modules (e.g., classes or packages). One programmer can say "I think an Observer would work great for publishing events from our stock market widget", and another programmers will know what that means. It becomes a shorthand for standard approaches.

Remember that design patterns are design patterns. They operate at a higher level than packages and frameworks, which frequently contain specific implementations of patterns.

A real bonus when programming in Python is that many design patterns are already built into the language. Examples of these are Singletons (AKA modules) and Decorators.

Types of patterns

- Four categories of patterns (or so)
    - Creational
    - Structural
    - Behavioral
    - Concurrency

There are four general categories of patterns, organized by how they are used within a project.

Creational patterns are used for software that creates objects and other entities. Well-known creational patterns include Abstract Factory and Singleton.

Structural patterns are used to create relationships between software components. Common structural patterns are Adapter, Proxy, and Flyweight.

Behavioral patterns are used to interact between components. Common behavioral patterns are Observer, Command, and Visitor.

Concurrency patterns are used to help coordinate between components used in multiprogramming. Examples are Double-checked locking, Object Pool, and Active object.

These are not the only ways to organize design patterns. Others have created more categories, or organized them in a totally different hierarchy.

Table 26. Creational Patterns

Pattern	Description
Abstract factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Builder	Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.
Factory method	Define an interface for creating a single object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses (dependency injection[15]).
Lazy initialization	Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. This pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the Proxy pattern.
Multiton	Ensure a class has only named instances, and provide a global point of access to them.
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalization of connection pool and thread pool patterns.
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
Resource acquisition	Ensure that resources are properly released by tying them to the lifespan of suitable objects.
Singleton	Ensure a class has only one instance, and provide a global point of access to it

Table 27. Structural Patterns

Pattern	**Description**
Adapter or Wrapper or Translator	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.
Bridge	Decouple an abstraction from its implementation allowing the two to vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to sub-classing for extending functionality.
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Flyweight	Use sharing to support large numbers of similar objects efficiently.
Front Controller	The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.
Module	Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity.
Proxy	Provide a surrogate or placeholder for another object to control access to it.
Twin	Twin allows modeling of multiple inheritance in programming languages that do not support this feature.

Table 28. Behavioral Patterns

Pattern	**Description**
Blackboard	Artificial intelligence pattern for combining disparate sources of data (see blackboard system)
Chain of responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
Memento	Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.
Null object	Avoid null references by providing a default object.
Observer or Publish/subscribe	Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.
Servant	Define common functionality for a group of classes.
Specification	Recombinable business logic in a Boolean fashion.
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Template method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Table 29. Concurrency Patterns

Pattern	Description
Active Object	Decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.
Balking	Only execute an action on an object when the object is in a particular state.
Binding properties	Combining multiple observers to force properties in different objects to be synchronized or coordinated in some way.[19]
Block chain	Decentralized way to store data and agree on ways of processing it in a Merkle tree, optionally using Digital signature for any individual contributions.
Double-checked locking	Reduce the overhead of acquiring a lock by first testing the locking criterion (the 'lock hint') in an unsafe manner; only if that succeeds does the actual locking logic proceed.
Event-based asynchronous	Addresses problems with the asynchronous pattern that occur in multi-threaded programs.[20]
Guarded suspension	Manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.
Join	Join-pattern provides a way to write concurrent, parallel and distributed programs by message passing. Compared to the use of threads and locks, this is a high level programming model.
Lock	One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it.[21]
Messaging design pattern (MDP)	Allows the interchange of information (i.e. messages) between components and applications.
Monitor object	An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time.
Reactor	A reactor object provides an asynchronous interface to resources that must be handled synchronously.
Read-write lock	Allows concurrent read access to an object, but requires exclusive access for write operations.
Scheduler	Explicitly control when threads may execute single-threaded code.
Thread pool	A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Can be considered a special case of the object pool pattern.
Thread-specific storage	Static or "global" memory local to a thread.

Singleton

- Ensures only one instance is created
- Singleton class requires metaprogramming in Python
- Variation called the Borg
- A Python module is already a singleton

The Singleton pattern is used to provide global access to a resource, when it is expensive, or incorrect, to have multiple copies available. The singleton can also control concurrent access.

Typical use cases for the singleton are loggers, spoolers, etc., as well as database or network connections.

There are three standard ways to implement Single classes in Python: using a module (no classes involved), a "classic" singleton, or a Borg.

Module as Singleton

- Modules are only loaded once
- Global variables in module are available after import

Since a python module is already, by design, a singleton. Rather than creating a Singleton class, you can just put data and methods in a normal module, and the module can be imported from anywhere in the application. The data and methods will only exist in one place, and there will never be more than one instance. This is useful for simple cases, but it is limited in some ways.

Example

designpatterns/singletons/singletonmodule/dbconn.py

```python
#!/usr/bin/env python
# (c)2015 John Strickler

# only one connection and one cursor will be created

import sqlite3

db_connection = sqlite3.connect(':memory:')

db_cursor = db_connection.cursor()
```

Example

designpatterns/singletons/singletonmodule/singleton_main.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

from dbconn import db_cursor
from builddb import build_database

build_database()

db_cursor.execute(
    '''
        select first_name, last_name
        from computer_people
    '''
)

for row in db_cursor.fetchall():
    print(' '.join(row))
```

designpatterns/singletons/singletonmodule/singleton_main.py

```
Bill Gates
Steve Jobs
Paul Allen
Larry Ellison
Mark Zuckerberg
Sergey Brin
Larry Page
Linux Torvalds
```

Classic Singleton

- Override *new*
- *init* is still called for each instance

To create a classic singleton, override the *new* method of a class. This method is normally not used in creating classes, because the *init*() takes the role of constructor, but *new*() is the "real" constructor, in that it returns the actual new object when an instance is requested.

One thing to be careful about is that *init*() is still called for each instance, so put a test in *init*() to make sure you're only initializing the instance once.

Example

designpatterns/singletons/singletonclassic/databaseconnection.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

# only one connection and one cursor will be created

import sqlite3

class DatabaseConnection(object):
    def __new__(cls):
        if not hasattr(cls, 'instance'):
            cls.instance = super(DatabaseConnection, cls).__new__(cls)
        return cls.instance

    def __init__(self):
        # IMPORTANT! One-time setup
        if not hasattr(self, '_db_connection'):
            self._db_connection = sqlite3.connect(':memory:')
            self._db_cursor = self._db_connection.cursor()

    @property
    def cursor(self):
        return self._db_cursor
```

Example

designpatterns/singletons/singletonclassic/builddb.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

# even though we are importing DatabaseConnection, it does not
# create a new connection
from databaseconnection import DatabaseConnection

db_conn = DatabaseConnection()  # get the instance

PEOPLE = [
    ('Bill', 'Gates', 'Microsoft'),
    ('Steve', 'Jobs', 'Apple'),
    ('Paul', 'Allen', 'Microsoft'),
    ('Larry', 'Ellison', 'Oracle'),
    ('Mark', 'Zuckerberg', 'Facebook'),
    ('Sergey','Brin', 'Google'),
    ('Larry', 'Page', 'Google'),
    ('Linux', 'Torvalds', 'Linux'),
]

def build_database():
    db_conn.cursor.execute('''
        create table computer_people (
            first_name varchar(16),
            last_name varchar(16),
            known_for varchar(16)
        );
    ''')

    insert_query = '''
        insert into computer_people
        (first_name, last_name, known_for)
        values (?, ?, ?);
    '''

    db_conn.cursor.executemany(insert_query, PEOPLE)
```

Example

designpatterns/singletons/singletonclassic/singleton_main.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

from databaseconnection import DatabaseConnection

from builddb import build_database

build_database()

db_conn = DatabaseConnection()

db_conn.cursor.execute(
    '''
        select first_name, last_name
        from computer_people
    '''
)

for row in db_conn.cursor.fetchall():
    print(' '.join(row))
```

designpatterns/singletons/singletonclassic/singleton_main.py

```
Bill Gates
Steve Jobs
Paul Allen
Larry Ellison
Mark Zuckerberg
Sergey Brin
Larry Page
Linux Torvalds
```

The Borg

- Singleton that creates multiple instances
- BUT, each instance shares same state with all others
- Allows inheritance from singleton
- AKA monostate

If you don’t need to inherit from a Singleton, then the classic implementation is good. However, when you need to inherit, you can use the Borg implementation, named for the alien race on Star Trek TNG.

In the Borg, each instance is a distinct object, but all of the instances share the same state.

This is done by sharing data dictionaries at the class level. The *new*() method returns a new instance object, but replaces the default dictionary with the shared one. Thus, all Borg instances really share the same set of attributes.

Example

designpatterns/singletons/singletonborg/databaseconnection.py

```python
#!/usr/bin/env python
# (c)2015 John Strickler

import sqlite3

class DatabaseConnection(object):
    _shared = {}

    def __new__(cls, *args, **kwargs):
        instance = super(DatabaseConnection, cls).__new__(cls, *args, **kwargs)

        if not '_db_connection' in cls._shared:
            db_conn = sqlite3.connect(':memory:')
            db_cursor = db_conn.cursor()

            cls._shared['_db_connection'] = db_conn
            cls._shared['_db_cursor'] = db_cursor

        instance.__dict__ = cls._shared


        return instance

    @property
    def cursor(self):
        return self._db_cursor
```

Example

designpatterns/singletons/singletonborg/builddb.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

# even though we are importing DatabaseConnection, it does not
# create a new connection
from databaseconnection import DatabaseConnection

db_conn = DatabaseConnection()  # get the instance

PEOPLE = [
    ('Bill', 'Gates', 'Microsoft'),
    ('Steve', 'Jobs', 'Apple'),
    ('Paul', 'Allen', 'Microsoft'),
    ('Larry', 'Ellison', 'Oracle'),
    ('Mark', 'Zuckerberg', 'Facebook'),
    ('Sergey','Brin', 'Google'),
    ('Larry', 'Page', 'Google'),
    ('Linux', 'Torvalds', 'Linux'),
]

def build_database():
    db_conn.cursor.execute('''
        create table computer_people (
            first_name varchar(16),
            last_name varchar(16),
            known_for varchar(16)
        );
    ''')

    insert_query = '''
        insert into computer_people
        (first_name, last_name, known_for)
        values (?, ?, ?);
    '''

    db_conn.cursor.executemany(insert_query, PEOPLE)
```

Example

designpatterns/singletons/singletonborg/singleton_main.py

```python
#!/usr/bin/env python
# (c)2015 John Strickler

from databaseconnection import DatabaseConnection

from builddb import build_database

build_database()

db_conn = DatabaseConnection()

db_conn.cursor.execute(
    '''
        select first_name, last_name
        from computer_people
    '''
)

for row in db_conn.cursor.fetchall():
    print(' '.join(row))
```

designpatterns/singletons/singletonborg/singleton_main.py

```
Bill Gates
Steve Jobs
Paul Allen
Larry Ellison
Mark Zuckerberg
Sergey Brin
Larry Page
Linux Torvalds
```

Strategy

- Encapsulate algorithms to make them interchangeable
- Isolate the things that vary from the things that don't
- Provides an interface for behavior
- Encourages loose coupling

The Strategy pattern is used when you have behaviors that may vary, but the use of the behavior stays the same in a client.

The client may want to sort, but we may want to be able to change the sort algorithm independently, from quicksort to mergesort to Timsort. Strategy decouples the sorting algorithm from the component that wants to sort.

The client gets and instance of the algorithm component, which is typically a class written to an API (an interface, in some languages). As long as the class provides the correct API, the client doesn't care about the algorithm's details.

Another example is saving a file from a word processor. The interface can select a class that saves in native format, XML, PDF, plain text, or translates to Klingon. All it has to do is get an instance of NativeWriter, XMLWriter, PDFWriter, PlainTextWriter, or KlingonWriter. No matter what kind of writer, it just passes the current document to the writer's write() method. Of course in this case the writer component will need to be able to understand the in-memory document format.

Example

designpatterns/strategies/simuduck/flylib.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

from abc import ABCMeta, abstractmethod

class FlyBase(object, metaclass=ABCMeta):
    @abstractmethod
    def fly(self):
        pass

class FlyWithWings(FlyBase):
    def fly(self):
        print("Soaring on my wings")

class Flightless(FlyBase):
    def fly(self):
        print("Gee, I can't fly")

if __name__ == '__main__':
    fww = FlyWithWings()
    fww.fly()

    fl = Flightless()
    fl.fly()
```

Example

designpatterns/strategies/simuduck/quacklib.py

```
#!/usr/bin/env python
# (c)2015 John Strickler
from abc import ABCMeta, abstractmethod

class QuackBase(object, metaclass=ABCMeta):
    @abstractmethod
    def quack(self):
        pass

class Quack(QuackBase):
    def quack(self):
        print('"Quack, quack"')

class Squeak(QuackBase):
    def quack(self):
        print('"Squeaky-squeaky"')

class MuteQuack(QuackBase):
    def quack(self):
        print("<I can't make any sounds>")

if __name__ == '__main__':
    q = Quack()
    q.quack()

    s = Squeak()
    s.quack()

    m = MuteQuack()
    m.quack()
```

Example

designpatterns/strategies/simuduck/duck.py

```python
#!/usr/bin/env python
# (c)2015 John Strickler
import flylib
import quacklib

class Duck(object):

    def __init__(
            self,
            duck_type,
            quack=None,
            fly=None
    ):
        self._duck_type = duck_type
        self._quack_behavior = quacklib.Quack() if quack is None else quack
        self._fly_behavior = flylib.FlyWithWings() if fly is None else fly


    def swim(self):
        print("Swimming")

    def display(self):
        print("I am a", self._duck_type)

    def set_quack(self, quack):
        if not issubclass(quack, quacklib.QuackBase):
            raise TypeError("Invalid quack")
        self._quack_behavior = quack

    def set_fly(self, fly):
        if not issubclass(fly, flylib.FlyBase):
            raise TypeError("Invalid flight type")
        self._fly_behavior = fly

    def quack(self):
        self._quack_behavior.quack()

    def fly(self):
        self._fly_behavior.fly()
```

Example

designpatterns/strategies/simuduck/simuduck_main.py

```python
#!/usr/bin/env python
# (c)2015 John Strickler

from duck import Duck
from quacklib import MuteQuack, Squeak
from flylib import Flightless

d1 = Duck('Mallard')
d2 = Duck('Robot', quack=MuteQuack())
d3 = Duck('Rubber Duckie', quack=Squeak(), fly=Flightless())

for d in d1, d2, d3:
    d.display()
    d.quack()
    d.fly()
    print('-' * 60)
```

designpatterns/strategies/simuduck/simuduck_main.py

```
I am a Mallard
"Quack, quack"
Soaring on my wings
------------------------------------------------------------
I am a Robot
<I can't make any sounds>
Soaring on my wings
------------------------------------------------------------
I am a Rubber Duckie
"Squeaky-squeaky"
Gee, I can't fly
------------------------------------------------------------
```

Decorators

- Built into Python
- Implemented via functions or classes
- Can decorate functions or classes
- Can take parameters (but not required to)
- functools.wraps() preserves function’s properties

In Python, decorators are a special case. Not only are decorators provided by the standard library, such as property() or classmethod(), but they have a special syntax. The @ sign is used to apply a decorator to a function or class.

A decorator is a component that modifies some other component. The purpose is typically to add functionality, but there are no real restrictions on what a decorator can do. Many decorators register a component with some other component. For instance, the @app.route() decorator in Flask maps a URL to a view function.

Table 30. Decorators in the standard library

Decorator	Description
`@abc.abstractmethod`	Indicate abstract method (must be implemented).
`@abc.abstractproperty`	Indicate abstract property (must be implemented). **DEPRECATED**
`@asyncio.coroutine`	Mark generator-based coroutine.
`@atexit.register`	Register function to be executed when interpreter (script) exits.
`@classmethod`	Indicate class method (receives class object, not instance object)
`@contextlib.contextmanager`	Define factory function for **with** statement context managers (no need to create __enter__() and __exit__() methods)
`@functools.lru_cache`	Wrap a function with a memoizing callable
`@functools.singledispatch`	Transform function into a single-dispatch generic function.
`@functools.total_ordering`	Supply all other comparison methods if class defines at least one.
`@functools.wraps`	Invoke update_wrapper() so decorator's replacement function keeps original function's name and other properties.
`@property`	Indicate a class property.
`@staticmethod`	Indicate static method (passed neither instance nor class object).
`@types.coroutine`	Transform generator function into a coroutine function.
`@unittest.mock.patch`	Patch target with a new object. When the function/with statement exits patch is undone.
`@unittest.mock.patch.dict`	Patch dictionary (or dictionary-like object), then restore to original state after test.
`@unittest.mock.patch.multiple`	Perform multiple patches in one call.
`@unittest.mock.patch.object`	Patch object attribute with mock object.
`@unittest.skip()`	Skip test unconditionally
`@unittest.skipIf()`	Skip test if condition is true
`@unittest.skipUnless()`	Skip test unless condition is true
`@unittest.expectedFailure()`	Mark Test as expected failure
`@unittest.removeHandler()`	Remove Control-C handler

Using decorators

- Provide a wrapper around a function or class
- Syntax

```
@decorator
def function():
    function_body
```

A decorator is a function or class that acts as a wrapper around a function or class. It allows you to modify the target without changing the target itself.

The @property, @classmethod, and @staticmethod decorators were described in the chapter on OOP.

Table 31. Decorator implementations

Implemented as (wrapper)	Decorates (target)	Takes params	Implementation [1]	Example as function call (without @)
function	function	No	Decorator returns replacement	`target = decorator(target)`
function	function	Yes	Decorator returns wrapper function Wrapper returns replacement function	`target = decorator(params)(target)`
class	function	No	`__call__` ***IS*** replacement function	`target = decorator(target)`
class	function	Yes	`__call__` ***RETURNS*** replacement function	`target = decorator(params)(target)`
function	class	No	Decorator modifies and returns original class	`target = decorator(target)`
function	class	Yes	Decorator returns wrapper Wrapper modifies and returns original class	`target = decorator(params)(target)`
class	class	No	`__new__` returns original class	`target = decorator(target)`
class	class	Yes	`__call__` returns original class	`target = decorator(params)(target)`

[1]In all cases, target is decorated with @decorator or @decorator(params). The example in column 5 is equivalent to the @ syntax, but is not normally used.

NOTE See the file **EXAMPLES/decorators/decorama.py** for examples of the above 8 different decorator implementations.

Creating decorator functions

- Decorator function gets original function
- Use functools.wraps

A decorator function is passed the target object (function or class), and returns a replacement object.

A simple decorator function expects only one argument – the function to be modified. It should return a new function, which will replace the original. The replacement function typically calls the original function as well as some new code.

A decorated function, like

```
@mydecorator
def myfunction():
    pass
```

is really called like this

```
myfunction = mydecorator(myfunction)
```

The new function should be defined with generic arguments so it can handle the original function's arguments.

The replacement function should be decorated with functools wraps. This makes sure the replacement function keeps the name (and other attributes) of the original function.

Example

designpatterns/decorators/deco_print_name.py

```
#!/usr/bin/env python

from functools import wraps

def print_name( old_func ):

    @wraps(old_func)
    def new_func( *args, **kwargs ):
        # added functionality
        print("==> Calling function {0}".format(old_func.__name__))
        return old_func( *args, **kwargs )  # call the 'real' function

    return new_func   # return the new function object


@print_name
def hello():
    print("Hello!")

@print_name
def goodbye():
    print("Goodbye!")

hello()
goodbye()
hello()
goodbye()
```

designpatterns/decorators/deco_print_name.py

```
==> Calling function hello
Hello!
==> Calling function goodbye
Goodbye!
==> Calling function hello
Hello!
==> Calling function goodbye
Goodbye!
```

Decorators with arguments

If the decorator itself needs arguments, then things get a little bit trickier. The decorator function gets the arguments, and returns a wrapper function that gets the original function (the one being decorated), and the wrapper function then returns the replacement function.

That is,

```
@mydecorator(param)
def myfunction():
    pass
```

is really called like this

```
myfunction = mydecorator(param)(myfunction)
```

Example

designpatterns/decorators/deco_print_name_with_label.py

```python
#!/usr/bin/env python

from functools import wraps

def print_name(label):

    def wrapper(old_func):

        @wraps(old_func)
        def new_func( *args, **kwargs ):
            # added functionality
            print("{0}: function {1}".format(
                label,
                old_func.__name__
            ))
            return old_func( *args, **kwargs )  # call the 'real' function

        return new_func   # return the new function object
    return wrapper

@print_name('HELLO')
def hello():
    print("Hello!")

@print_name('HELLO')
def howdy():
    print("Howdy!")

@print_name('GOODBYE')
def goodbye():
    print("Goodbye!")

@print_name('GOODBYE')
def solong():
    print("So long!")


hello()
howdy()
goodbye()
solong()
```

designpatterns/decorators/deco_print_name_with_label.py

```
HELLO: function hello
Hello!
HELLO: function howdy
Howdy!
GOODBYE: function goodbye
Goodbye!
GOODBYE: function solong
So long!
```

Adapters

- Help make two incompatible interfaces compatible
- Prevent making changes to either interface
- Work with objects after implementation
- Adapt one interface to the other

The Adapter pattern is used to make two incompatible interfaces compatible. This is needed when you are not able to change the software on one side or the other. For instance, you're using a third-party library that you can't change. Your software was written for an older version of the library, and when they update the library, suddenly your code is broken. While in the ideal world, you could update the actual component that needs the library, an Adapter can let your existing code work with the updated library.

It can also be used for data conversions. Let's say you're getting data from a web service, but the units are kilometers and you need them in miles. You can create an adapter that connects to the original service, and converts the data before returning it to the requesting component.

An Adapter takes an instance of the caller, and uses the information in the instance to make the appropriate calls to the callee.

In the Java library, the IO readers and writers are adapters.

A Proxy is similar to an Adapter, but the Proxy presents the same interface as the component it represents, while an Adapter does not.

Example

designpatterns/adapters/cat_adapter.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

class Cat(object):
    def meow(self):
        print("Meow!!")

class Dog(object):
    def bark(self):
        print("Arf arf!!")

class CatAdapter(Dog):
    def __init__(self, cat):
        self._cat = cat

    def bark(self):
        self._cat.meow()

Garfield = Cat()

dog = CatAdapter(Garfield)
dog.bark()
```

designpatterns/adapters/cat_adapter.py

```
Meow!!
```

Abstract Factory

- Create families of objects
- Meta-factory (factory of factories)
- Provide various objects that share API
- Alias: Kit
- Use cases: – System should be independent of how products are created – System configured with one of multiple families of products – Enforce constraints on family of related products – Create abstract classes for product family

Participating objects: Abstract Factory, Factory, Abstract Product, Product, Client

An Abstract Factory is a creational pattern that is useful when you want to decouple the creation of classes from an application (or other component).

An Abstract Factory can be considered a meta-factory in that it typically returns a specific factory which creates the desired object. The client code passes a parameter to the Abstract Factory to tell it which factory to return.

While a Factory uses inheritance, an Abstract Factory uses composition.

The official GoF definition of the Abstract Factory is:

“Provide an interface for creating families of related or dependent objects without specifying their concrete classes.”

Builder

- Separate construction from representation
- Creates object composed of multiple parts
- Object not complete until all parts created
- Same builder can create multiple representations
- Use cases
    - Product creation algorithm should be independent of parts and assembly
    - Construction process should allow different representations of product

Participating objects: Builder Mixin, Builder, Director, Product

A Builder is a creational pattern that separates the construction of a product from its representation. A builder object creates the multiple parts of the target object. A director object controls the building of an object by using an instance of a builder object.

The representation of the product can vary, but the process remains the same.

Example

builder_pattern.py

```
from abc import ABCMeta, abstractmethod


class Director(object):

    def construct(self, builder):
        '''
        Builder uses multiple steps
        :param builder: A Builder object
        :return:
        '''
        builder.build_part_A()
        builder.build_part_B()

class BuilderBase(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def build_part_A(self): pass

    @abstractmethod
    def build_part_B(self): pass

    @abstractmethod
    def get_result(self): pass

class Product(object):
    def __init__(self, name):
        self._name = name
        self._parts = []

    def add(self, part):
        self._parts.append(part)

    def show(self):
        print("{} Parts -------".format(self._name))
        for part in self._parts:
            print(part)
        print()

class Builder1(BuilderBase):
    def __init__(self):
        self._product = Product("Product 1")
    def build_part_A(self):
        self._product.add("PartA")

    def build_part_B(self):
        self._product.add("PartB")

    def get_result(self):
        return self._product


class Builder2(BuilderBase):
    def __init__(self):
        self._product = Product("Product 2")

    def build_part_A(self):
        self._product.add("PartX")

    def build_part_B(self):
        self._product.add("PartY")

    def get_result(self):
        return self._product

if __name__ == '__main__':
    director = Director()
    builder1 = Builder1()
    builder2 = Builder2()

    director.construct(builder1)
    product1 = builder1.get_result()
    product1.show()

    director.construct(builder2)
    product2 = builder2.get_result()
    product2.show()
```

builder_pattern.py

```
Product 1 Parts -------
PartA
PartB

Product 2 Parts -------
PartX
PartY
```

Facade

- Simplified front end to a system
- Hides the details of a complex component
- Implemented as class or function
- Use cases – Simple interface for complex subsystem – Decouple subsystem from other subsystems – Create layered subsystems

Participating objects: Facade, subsystem classes

A Facade is a design pattern that creates a simplified interface for a complex component. It essentially hides the details of a system.

For instance, the requests module is a Facade for the urllib.* packages. requests makes proxies, authentication, and sending data much easier than doing everything individually.

Another name for a Facade could be "wrapper". It "wraps" some other component. One downside to a Facade is that it might not provide access to all the features available in the target component. A Facade provides a new interface to the component.

Example

csv_facade.py

```python
#
"""
Demo of the Facade design pattern
"""
import csv

PRESIDENTS_FILE = '../DATA/presidents.csv'

class CSVUpper():
    """
    A facade for a CSV file
    """
    def __init__(self, csv_file):
        self._file_in = open(csv_file)
        self._rdr = csv.reader(self._file_in)

    def __iter__(self):
        return self

    def __next__(self):
        row = next(self._rdr)
        if row:
            return [r.upper() for r in row]
        else:
            raise StopIteration()

    def __del__(self):
        self._file_in.close()

if __name__ == '__main__':
    presidents = CSVUpper(PRESIDENTS_FILE)
    for p in presidents:
        print(p)
```

csv_facade.py

```
['1', 'GEORGE', 'WASHINGTON', 'WESTMORELAND COUNTY', 'VIRGINIA', 'NO PARTY']
['2', 'JOHN', 'ADAMS', 'BRAINTREE, NORFOLK', 'MASSACHUSETTS', 'FEDERALIST']
['3', 'THOMAS', 'JEFFERSON', 'ALBERMARLE COUNTY', 'VIRGINIA', 'DEMOCRATIC - REPUBLICAN']
['4', 'JAMES', 'MADISON', 'PORT CONWAY', 'VIRGINIA', 'DEMOCRATIC - REPUBLICAN']
['5', 'JAMES', 'MONROE', 'WESTMORELAND COUNTY', 'VIRGINIA', 'DEMOCRATIC - REPUBLICAN']
['6', 'JOHN QUINCY', 'ADAMS', 'BRAINTREE, NORFOLK', 'MASSACHUSETTS', 'DEMOCRATIC -
REPUBLICAN']
['7', 'ANDREW', 'JACKSON', 'WAXHAW', 'SOUTH CAROLINA', 'DEMOCRATIC']
['8', 'MARTIN', 'VAN BUREN', 'KINDERHOOK', 'NEW YORK', 'DEMOCRATIC']
['9', 'WILLIAM HENRY', 'HARRISON', 'BERKELEY', 'VIRGINIA', 'WHIG']
['10', 'JOHN', 'TYLER', 'CHARLES CITY COUNTY', 'VIRGINIA', 'WHIG']
['11', 'JAMES KNOX', 'POLK', 'MECKLENBURG COUNTY', 'NORTH CAROLINA', 'DEMOCRATIC']
['12', 'ZACHARY', 'TAYLOR', 'ORANGE COUNTY', 'VIRGINIA', 'WHIG']
['13', 'MILLARD', 'FILLMORE', 'CAYUGA COUNTY', 'NEW YORK', 'WHIG']
['14', 'FRANKLIN', 'PIERCE', 'HILLSBORO', 'NEW HAMPSHIRE', 'DEMOCRATIC']
['15', 'JAMES', 'BUCHANAN', 'COVE GAP', 'PENNSYLVANIA', 'DEMOCRATIC']
['16', 'ABRAHAM', 'LINCOLN', 'HODGENVILLE, HARDIN COUNTY', 'KENTUCKY', 'REPUBLICAN']
['17', 'ANDREW', 'JOHNSON', 'RALEIGH', 'NORTH CAROLINA', 'REPUBLICAN']
['18', 'ULYSSES SIMPSON', 'GRANT', 'POINT PLEASANT', 'OHIO', 'REPUBLICAN']
['19', 'RUTHERFORD BIRCHARD', 'HAYES', 'DELAWARE', 'OHIO', 'REPUBLICAN']
['20', 'JAMES ABRAM', 'GARFIELD', 'ORANGE, CUYAHOGA COUNTY', 'OHIO', 'REPUBLICAN']
```

Example

remote_cmd_facade.py

```python
import paramiko
REMOTE_HOST = 'localhost'
REMOTE_USER = 'python'
REMOTE_PASSWORD = 'l0lz'

class RemoteConnection():
    def __init__(self, host, user, password):
        self._host = host
        self._user = user
        self._password = password
        self._connect()

    def _connect(self):
        self._ssh = paramiko.SSHClient()
        self._ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        self._ssh.connect(self._host, username=self._user, password=self._password)

    def run_command(self, command_line):
        stdin, stdout, stderr = self._ssh.exec_command(command_line)
        stdout_text = stdout.read()
        stderr_text = stderr.read()
        return stdout_text, stderr_text

    def __del__(self):
        self._ssh.close()

if __name__ == '__main__':
    rc = RemoteConnection(REMOTE_HOST, REMOTE_USER, REMOTE_PASSWORD)

    stdout, stderr = rc.run_command('whoami')
    print(stdout.decode(), '\n')

    stdout, stderr = rc.run_command('grep root /etc/passwd')
    print("STDOUT:", stdout.decode())
    print("STDERR:", stderr.decode())

    print('-' * 60)

    stdout, stderr = rc.run_command('grep root /etc/pizza')
    print("STDOUT:", stdout.decode())
    print("STDERR:", stderr.decode())
```

remote_cmd_facade.py

```
python



STDOUT: root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
_cvmsroot:*:212:212:CVMS Root:/var/empty:/usr/bin/false

STDERR:
------------------------------------------------------------
STDOUT:
STDERR: grep: /etc/pizza: No such file or directory
```

Flyweight

- Many small objects that share state
- E.g. characters in a word processor
- Use cases
    - Applications that need a large number of objects
    - Number of objects would drive up storage costs
    - Object state can be easily made external
    - Factor out duplicate state among many objects
    - Application doesn’t require unique objects

Participating objects: Flyweight Mixin, Flyweight, Unshared Flyweight, Flyweight Factory, Client

While it is useful to have objects at a fundamental level of granularity, it can be expensive in terms of memory usage, and possibly performance.

As an example, consider a word processor. It would be nice to represent individual characters as objects, but it would take large amounts of memory if there were separate objects for every character in a large document.

The Flyweight pattern allows you to create many instances of an objects, but instances can share state. In the word processor example, all instances of the letter "m" would really be the same object. Any context-dependent information is calculated by or passed in from the clients (i.e., objects using the FlyWeight objects.

There are several ways to implement a Flyweight in Python. In all cases, there is a repository of instances, and instances are only created via a factory. In the example, a decorator is used to turn a class into a Flyweight, and the class itself is the factory via the *call*method.

Each time an instance is needed, it is either drawn from the dictionary of instances, or created and added to the dictionary. The dictionary acts as a cache of the state of all instances.

NOTE For another example of using the FlyWeight decorator, see bigcat.py in the EXAMPLES folder

Command

- Encapsulate an operation
- Decoupled from the invoker
- Commands can be grouped
- Use cases – Parameter objects by task – Queue and execute tasks at different times – Support undo operations – Log changes to a system for replay – Transaction-based systems

Participating objects Command Mixin, Command, Client, Invoker, Receiver

The command pattern encapsulates an operation (or request). The code that executes the command is known as the invoker, and does not have to know the implementation details of the command object; it only needs to know the API, which can be inherited from an abstract base class to make sure the command object implements the necessary methods.

Individual command objects may be queued, and then executed in order at a specified time.

Example

command_pattern.py

```
#
from abc import ABCMeta, abstractmethod

class Command(metaclass=ABCMeta):
    '''Command "interface"'''

    @abstractmethod
    def execute(self): pass


class Fan:

    def start_rotate(self):
        print("Fan is rotating")

    def stop_rotate(self):
        print("Fan is not rotating")


class Light:

    def turn_on(self):
        print("Light is on")


    def turn_off(self):
        print("Light is off")


class Switch:

    def __init__(self, on, off):
        self.on_command = on
        self.off_command = off

    def on(self):
        self.on_command.execute()

    def off(self):
        self.off_command.execute()


class LightOnCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_on()


class LightOffCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_off()


class FanOnCommand(Command):
    def __init__(self, fan):
        self.fan = fan

    def execute(self):
        self.fan.start_rotate()

class FanOffCommand(Command):
    def __init__(self, fan):
        self.fan = fan

    def execute(self):
        self.fan.stop_rotate()

if __name__ == '__main__':
    light = Light()
    light_on_command = LightOnCommand(light)
    light_off_command = LightOffCommand(light)

    light_switch = Switch(light_on_command, light_off_command)
    light_switch.on()
    light_switch.off()

    fan = Fan()
    fan_on_command = FanOnCommand(fan)
    fan_off_command = FanOffCommand(fan)

    fan_switch = Switch(fan_on_command, fan_off_command)
    fan_switch.on()
    fan_switch.off()
```

command_pattern.py

```
Light is on
Light is off
Fan is rotating
Fan is not rotating
```

Mediator

- Controls interaction among related objects
- Similar to facade, but two-way
- Objects don’t interact directly
- Use cases – Organize complex interdependence of objects – Reuse an object that interacts with other objects – Configure multi-object behavior

Participants

```
Mediator Mixin, Mediator, Colleague Classes
```

A mediator encapsulates interaction among related classes. Mediators are similar to facades, but are two-way, where a facade is generally one-way. A mediator knows the behavior of multiple objects, so that every object doesn’t have to know about every other object. It reduces the coupling within a system that has many objects.

Chapter 19 Exercises

Exercise 19-1 (deco_timer.py)

Create a decorator named functiontimer that can be applied to a function and reports the number of seconds (fractional) it takes for the function to execute. You can get the time from the time.time() function. (i.e., import time, and call the time() function from it.)

Just subtract the start time from the end time and report it. You don’t need any further calculations.

Write some functions and test your decorator on them.

Index