Custom Intermediate Python

John Strickler

Version 1.0, February 2020: $

Table of Contents

About this course

Welcome!

- We're glad you're here
- Class has hands-on labs for nearly every chapter
- Please make a name tent

Instructor name:

Instructor e-mail:

Have Fun!

Classroom etiquette

- 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 classroom 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 killer rabbits to class. They might maim, dismember, or otherwise disturb your fellow students.

Course Outline

Day 1

Chapter 1 Functions, Modules, and Packages
Chapter 2 Intermediate Classes
Chapter 3 PyQt
Chapter 4 Regular Expressions
Chapter 5 Pythonic Programming

Optional topics

Chapter 6 Scripting for System Administration
Chapter 7 Working With Files
Chapter 8 Intro to Numpy
Chapter 9 Intro to Pandas
Chapter 10 Intro to Matplotlib
Chapter 11 Flask Overview
Chapter 12 Using SQLAlchemy with Flask
Chapter 13 Restful Services with Flask
Chapter 14 Network Programming
Chapter 15 Multiprogramming
Chapter 16 Serializing Data

NOTE Optional topics will be covered as time permits. It is not anticipated that there will be time for many of the optional topics.

Student files

You will need to load some 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 **py3bnpinterm**.

What's in the files?

py3bnpinterm contains data and other files needed for the exercises
py3bnpinterm/EXAMPLES contains the examples from the course manuals.
py3bnpinterm/ANSWERS contains sample answers to the labs.

WARNING	The student files do not contain Python itself. It will need to be installed separately. This has probably already been done for you.

Extracting the student files

Windows

Open the file **py3bnpinterm.zip**. Extract all files to your desktop. This will create the folder **py3bnpinterm**.

Non-Windows (includes Linux, OS X, etc)

Copy or download **py3bnpinterm.tgz** to your home directory. In your home directory, type

```
tar xzvf py3bnpinterm.tgz
```

This will create the **py3bnpinterm** directory under your home directory.

Examples

Nearly all examples from the course manual are provided in the EXAMPLES subdirectory.

It will look like this:

Example

cmd_line_args.py

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

import sys   ①

print(sys.argv) ②

name = sys.argv[1]  ③
print("name is", name)
```

① Import the **sys** module

② Print all parameters, including script itself

③ Get the first actual parameter

cmd_line_args.py Fred

```
['/Users/jstrick/curr/courses/python/examples3/cmd_line_args.py', 'Fred']
name is Fred
```

Lab Exercises

- Relax – the labs are not quizzes
- Feel free to modify labs
- Ask the instructor for help
- Work on your own scripts or data
- Answers are in py3bnpinterm/ANSWERS

Appendices

- Appendix A: Python Bibliography

Chapter 1: Functions, Modules and Packages

Objectives

- Define functions
- Learn the four kinds of function parameters
- Create new modules
- Load modules with **import**
- Set module search locations
- Organize modules into packages
- Alias module and package names

Functions

- Defined with **def**
- Accept parameters
- Return a value

Functions are a way of isolating code that is needed in more than one place, refactoring code to make it more modular. They are defined with the **def** statement.

Functions can take various types of parameters, as described on the following page. Parameter types are dynamic.

Functions can return one object of any type, using the **return** statement. If there is no return statement, the function returns **None**.

TIP Be sure to separate your business logic (data and calculations) from your presentation logic (the user interface).

Example

function_basics.py

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


def say_hello():  ①
    print("Hello, world")
    print()
    ②


say_hello()  ③


def get_hello():
    return "Hello, world"  ④


h = get_hello()  ⑤
print(h)
print()


def sqrt(num):  ⑥
    return num ** .5


m = sqrt(1234)  ⑦
n = sqrt(2)

print("m is {:.3f} n is {:.3f}".format(m, n))
```

① Function takes no parameters

② If no **return** statement, return None

③ Call function (arguments, if any, in ())

④ Function returns value

⑤ Store return value in h

⑥ Function takes exactly one argument

⑦ Call function with one argument

function_basics.py

```
Hello, world

Hello, world

m is 35.128 n is 1.414
```

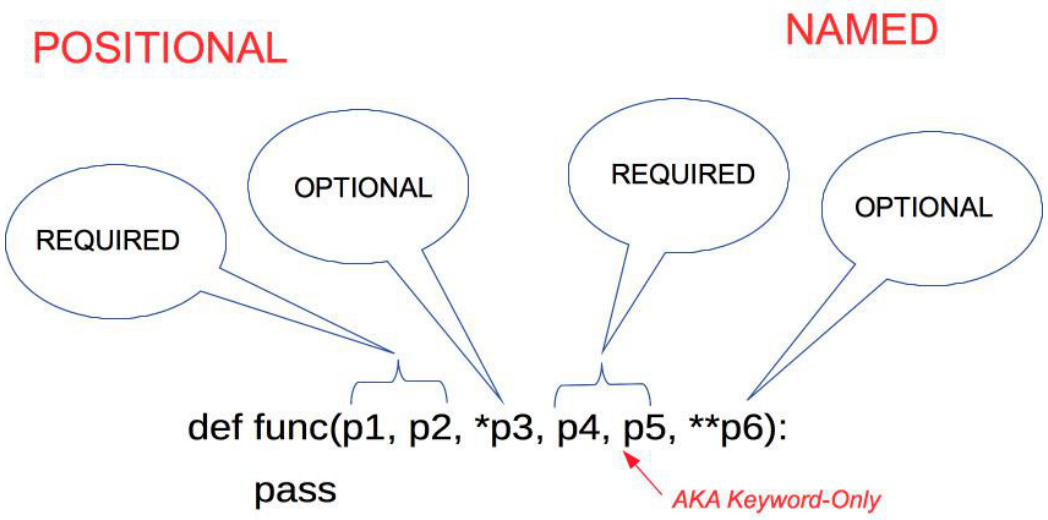
Function parameters

- Positional or named
- Required or optional
- Can have default values

Functions can accept both positional and named parameters. Furthermore, parameters can be required or optional. They must be specified in the order presented here.

The first set of parameters, if any, is a set of comma-separated names. These are all required. Next you can specify a variable preceded by an asterisk — this will accept any optional parameters.

After the optional positional parameters you can specify required named parameters. These must come after the optional parameters. If there are no optional parameters, you can use a plain asterisk as a placeholder. Finally, you can specify a variable preceded by two asterisks to accept optional named parameters.



Example

function_parameters.py

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

def fun_one():  ①
    print("Hello, world")


print("fun_one():", end=' ')
fun_one()
print()


def fun_two(n):  ②
    return n ** 2


x = fun_two(5)
print("fun_two(5) is {}\n".format(x))


def fun_three(count=3):  ③
    for _ in range(count):
        print("spam", end=' ')
    print()


fun_three()
fun_three(10)
print()


def fun_four(n, *opt):  ④
    print("fun_four():")
    print("n is ", n)
    print("opt is", opt)
    print('-' * 20)


fun_four('apple')
fun_four('apple', "blueberry", "peach", "cherry")
```

```
def fun_five(*, spam=0, eggs=0):  ⑤
    print("fun_five():")
    print("spam is:", spam)
    print("eggs is:", eggs)
    print()


fun_five(spam=1, eggs=2)
fun_five(eggs=2, spam=2)
fun_five(spam=1)
fun_five(eggs=2)
fun_five()


def fun_six(**named_args):  ⑥
    print("fun_six():")
    for name in named_args:
        print(name, "==> ", named_args[name])


fun_six(name="Lancelot", quest="Grail", color="red")
```

① no parameters

② one required parameter

③ one required parameter with default value

④ one fixed, plus optional parameters

⑤ keyword-only parameters

⑥ keyword (named) parameters

function_parameters.py

```
fun_one(): Hello, world

fun_two(5) is 25

spam spam spam
spam spam spam spam spam spam spam spam spam spam

fun_four():
n is  apple
opt is ()
--------------------
fun_four():
n is  apple
opt is ('blueberry', 'peach', 'cherry')
--------------------
fun_five():
spam is: 1
eggs is: 2

fun_five():
spam is: 2
eggs is: 2

fun_five():
spam is: 1
eggs is: 0

fun_five():
spam is: 0
eggs is: 2

fun_five():
spam is: 0
eggs is: 0

fun_six():
name ==>  Lancelot
quest ==>  Grail
color ==>  red
```

Default parameters

- Assigned with equals sign
- Used if no values passed to function

Required parameters can have default values. They are assigned to parameters with the equals sign. Parameters without defaults cannot be specified after parameters with defaults.

Example

default_parameters.py

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

def spam(greeting, whom='world'):  ①
    print("{}, {}".format(greeting, whom))


spam("Hello", "Mom")  ②
spam("Hello")  ③
print()

def ham(*, file_name, file_format='txt'):  ④
    print("Processing {} as {}".format(file_name, file_format))

ham(file_name='eggs')  ⑤
ham(file_name='toast', file_format='csv')
```

① 'world' is default value for positional parameter **whom**

② parameter supplied; default not used

③ parameter not supplied; default is used

④ 'world' is default value for named parameter **format**

⑤ parameter **format** not supplied; default is used

default_parameters.py

```
Hello, Mom
Hello, world

Processing eggs as txt
Processing toast as csv
```

Python Function parameter behavior (from PEP 3102)

For each formal parameter, there is a slot which will be used to contain the value of the argument assigned to that parameter.

- Slots which have had values assigned to them are marked as 'filled'. Slots which have no value assigned to them yet are considered 'empty'.
- Initially, all slots are marked as empty.
- Positional arguments are assigned first, followed by keyword arguments.
- For each positional argument:
  - Attempt to bind the argument to the first unfilled parameter slot. If the slot is not a vararg slot, then mark the slot as 'filled'.
  - If the next unfilled slot is a vararg slot, and it does not have a name, then it is an error.
  - Otherwise, if the next unfilled slot is a vararg slot then all remaining non-keyword arguments are placed into the vararg slot.
- For each keyword argument:
  - If there is a parameter with the same name as the keyword, then the argument value is assigned to that parameter slot. However, if the parameter slot is already filled, then that is an error.
  - Otherwise, if there is a 'keyword dictionary' argument, the argument is added to the dictionary using the keyword name as the dictionary key, unless there is already an entry with that key, in which case it is an error.
  - Otherwise, if there is no keyword dictionary, and no matching named parameter, then it is an error.
- Finally:
  - If the vararg slot is not yet filled, assign an empty tuple as its value.
  - For each remaining empty slot: if there is a default value for that slot, then fill the slot with the default value. If there is no default value, then it is an error.
- In accordance with the current Python implementation, any errors encountered will be signaled by raising TypeError.

Name resolution (AKA Scope)

- What is "scope"
- Scopes used dynamically
- Four levels of scope
- Assignments always go into the innermost scope (starting with local)

A scope is the area of a Python program where an unqualified (not preceded by a module name) name can be looked up.

Scopes are used dynamically. There are four nested scopes that are searched for names in the following order:

local	local names bound within a function
nonlocal	local names plus local names of outer function(s)
global	the current module's global names
builtin	built-in functions (contents of ***_builtins_*** module)

Within a function, all assignments and declarations create local names. All variables found outside of local scope (that is, outside of the function) are read-only.

Inside functions, local scope references the local names of the current function. Outside functions, local scope is the same as the global scope – the module's namespace. Class definitions also create a local scope.

Nested functions provide another scope. Code in function B which is defined inside function A has read-only access to all of A's variables. This is called **nonlocal** scope.

Example

scope_examples.py

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

x = 42  ①


def function_a():
    y = 5  ②

    def function_b():
        z = 32  ③
        print("function_b(): z is", z)  ④
        print("function_b(): y is", y)  ⑤
        print("function_b(): x is", x)  ⑥
        print("function_b(): type(x) is", type(x))  ⑦

    return function_b


f = function_a()  ⑧
f()  ⑨
```

① global variable

② local variable to function_a(), or nonlocal to function_b()

③ local variable

④ local scope

⑤ nested (nonlocal) scope

⑥ global scope

⑦ builtin scope

⑧ calling function_a, which returns function_b

⑨ calling function_b

scope_examples.py

```
function_b(): z is 32
function_b(): y is 5
function_b(): x is 42
function_b(): type(x) is <class 'int'>
```

The global statement

- global statement allows function to change globals
- nonlocal statement allows function to change nonlocals

The **global** keyword allows a function to modify a global variable. This is universally acknowledged as a BAD IDEA. Mutating global data can lead to all sorts of hard-to-diagnose bugs, because a function might change a global that affects some other part of the program. It's better to pass data into functions as parameters and return data as needed. Mutable objects, such as lists, sets, and dictionaries can be modified in-place.

The **nonlocal** keyword can be used like **global** to make nonlocal variables in an outer function writable.

Modules

- Files containing python code
- End with .py
- No real difference from scripts

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable *name*.

To use a module named spam.py, say `import spam`

This does not enter the names of the functions defined in spam directly into the symbol table; it only adds the module name **spam**. Use the module name to access the functions or other attributes.

Python uses modules to contain functions that can be loaded as needed by scripts. A simple module contains one or more functions; more complex modules can contain initialization code as well. Python classes are also implemented as modules.

A module is only loaded once, even there are multiple places in an application that import it.

Modules and packages should be documented with *docstrings*.

Using import

- import statement loads modules
- Three variations
  - import module
  - from module import function-list
  - from module import * use with caution!

There are three variations on the **import** statement:

Variation 1

import module
loads the module so its data and functions can be used, but does not put its attributes (names of classes, functions, and variables) into the current namespace.

Variation 2

from module import function, ...
imports only the function(s) specified into the current namespace. Other functions are not available (even though they are loaded into memory).

Variation 3

from module import *
loads the module, and imports all functions that do not start with an underscore into the current namespace. This should be used with caution, as it can pollute the current namespace and possibly overwrite builtin attributes or attributes from a different module.

NOTE

The first time a module is loaded, the interpreter creates a version compiled for faster loading. This version has platform information embedded in the name, and has the extension .pyc. These .pyc files are put in a folder named **__pycache__**.

Example

samplelib.py

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

# sample Python module


def spam():
    print("Hello from spam()")

def ham():
    print("Hello from ham()")

def _eggs():
    print("Hello from _eggs()")
```

use_samplelib1.py

```
#!/usr/bin/env python
import samplelib  ①

samplelib.spam()  ②
samplelib.ham()
```

① import samplelib module (samplelib.py) — creates object named **samplelib** of type "Module"

② call function spam() in module samplelib

use_samplelib1.py

```
Hello from spam()
Hello from ham()
```

use_samplelib2.py

```
#!/usr/bin/env python
from samplelib import spam, ham  ①

spam()  ②
ham()
```

① import functions spam and ham from samplelib module into current namespace — does not create the module object

② module name not needed to call function spam()

use_samplelib2.py

```
Hello from spam()
Hello from ham()
```

use_samplelib3.py

```
#!/usr/bin/env python
from samplelib import *  ①

spam()  ②
ham()
```

① import all functions (that do not start with _) from samplelib module into current namespace

② module name not needed to call function spam()

use_samplelib3.py

```
Hello from spam()
Hello from ham()
```

use_samplelib4.py

```
#!/usr/bin/env python
from samplelib import spam as pig, ham as hog  ①

pig()
hog()
```

① import functions spam and ham, aliased to pig and hog

use_samplelib4.py

```
Hello from spam()
Hello from ham()
```

How *import* * can be dangerous

- Imported names may overwrite existing names
- Be careful to read the documentation

Using `import *` to import all public names from a module has a bit of a risk. While generally harmless, there is the chance that you will unknowingly import a module that overwrites some previously-imported module.

To be 100% certain, always import the entire module, or else import names explicitly.

Example

electrical.py

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

default_amps = 10
default_voltage = 110
default_current = 'AC'

def amps():
    return default_amps

def voltage():
    return default_voltage

def current():
    return default_current
```

navigation.py

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

current_types = 'slow medium fast'.split()

def current():
    return current_types[0]
```

why_import_star_is_bad.py

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

from electrical import *  ①
from navigation import *  ②

print(current())  ③
print(voltage())
print(amps())
```

① import current *explicitly* from electrical

② import current *implicitly* from navigation

③ calls navigation.current(), not electrical.current()

why_import_star_is_bad.py

```
slow
110
10
```

how_to_avoid_import_star.py

```
#!/usr/bin/env python
import electrical as e  ①
import navigation as n  ②

print(e.current())  ③
print(n.current())  ④
```

how_to_avoid_import_star.py

```
AC
slow
```

Module search path

- Searches current folder first, then predefined locations
- Add custom locations to PYTHONPATH
- Paths stored in sys.path

When you specify a module to load with the import statement, it first looks in the current directory, and then searches the directories listed in sys.path.

```
>>> import sys
>>> sys.path
```

To add locations, put one or more directories to search in the PYTHONPATH environment variable. Separate multiple paths by semicolons for Windows, or colons for Unix/Linux. This will add them to **sys.path**, after the current folder, but before the predefined locations.

Windows

set PYTHONPATH=C:"\Documents and settings\Bob\Python"

Linux/OS X

export PYTHONPATH="/home/bob/python"

You can also append to sys.path in your scripts, but this can result in non-portable scripts, and scripts that will fail if the location of the imported modules changes.

```
import sys
sys.path.extend("/usr/dev/python/libs","/home/bob/pylib")
import module1
import module2
```

Executing modules as scripts

- *_name_* is current module.
  - set to **__main__** if run as script
  - set to module_name if imported
- test with `if name == "__main__"`
- Module can be run directly or imported

It is sometimes convenient to have a module also be a runnable script. This is handy for testing and debugging, and for providing modules that also can be used as standalone utilities.

Since the interpreter defines its own name as '__main__', you can test the current namespace's *name* attribute. If it is '__main__', then you are at the main (top) level of the interpreter, and your file is being run as a script; it was not loaded as a module.

Any code in a module that is not contained in function or method is executed when the module is imported.

This can include data assignments and other startup tasks, for example connecting to a database or opening a file.

Many modules do not need any initialization code.

Example

using_main.py

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


# other imports  (standard library, standard non-library, local)

# constants (AKA global variables)

# main function
def main(args):  ①
    function1()
    function2()


# other functions
def function1():
    print("hello from function1()")


def function2():
    print("hello from function2()")


if __name__ == '__main__':
    main(sys.argv[1:])  ②
```

① Program entry point. While **main** is not a reserved word, it is a strong convention

② Call main() with the command line parameters (omitting the script itself)

Packages

- Package is folder containing modules or packages
- Startup code goes in __init__.py (optional)

A package is a group of related modules or subpackages. The grouping is physical – a package is a folder that contains one or more modules. It is a way of giving a hierarchical structure to the module namespace so that all modules do not live in the same folder.

A package may have an initialization script named **__init__.py**. If present, this script is executed when the package or any of its contents are loaded. (In Python 2, **__init__.py** was required).

Modules in packages are accessed by prefixing the module with the package name, using the dot notation used to access module attributes.

Thus, if Module **eggs** is in package **spam**, to call the **scramble()** function in **eggs**, you would say `spam.eggs.scramble()`.

By default, importing a package name by itself has no effect; you must explicitly load the modules in the packages. You should usually import the module using its package name, like `from spam import eggs`, to import the eggs module from the spam package.

Packages can be nested.

Example

```
sound/                  Top-level package
    __init__.py         Initialize the sound package (optional)
    formats/            Subpackage for file formats
        __init__.py         Initialize the formats package (optional)
        wavread.py
        wavwrite.py
        aiffread.py
        aiffwrite.py
        auread.py
        auwrite.py
        ...
    effects/            Subpackage for sound effects
        __init__.py         Initialize the formats package (optional)
        echo.py
        surround.py
        reverse.py
        ...
    filters/            Subpackage for filters
        __init__.py         Initialize the formats package (optional)
        equalizer.py
```

```
from sound.formats import aiffread
import sound.effects
import sound.filters.equalizer
```

Configuring import with __init__.py

- load modules into package's namespace
- specify modules to load when * is used

For convenience, you can put import statements in a package's **__init__.py** to autoload the modules into the package namespace, so that import PKG imports all the (or just selected) modules in the package.

__init__.py can also be used to setup data or other resources that will be used by multiple modules within a package.

If the variable ***_all_*** in **__init__.py** is set to a list of module names, then only these modules will be loaded when the import is

```
from PKG import *
```

Given the following package and module layout, the table on the next page describes how **__init__.py** affects imports.

```
my_package
   |------__init__.py
   |------module_a.py
   |         function_a()
   |------module_b.py
   |         function_b()
   |------module_c.py
             function_c()
```

Import statement	What it does
If **__init__.py** is empty	
`import my_package`	Imports **my_package** only, but not contents. No modules are imported. This is not useful.
`import my_package.module_a`	Imports **module_a** into **my_package** namespace. Objects in **module_a** must be prefixed with **my_package.module_a**
`from my_package import module_a`	Imports **module_a** into main namespace. Objects in **module_a** must be prefixed with **module_a**
`from my_package import module_a, module_b`	Imports **module_a** and **module_b** into main namespace.
`from my_package import *`	Does not import anything!
`from my_package.module_a import *`	Imports all contents of **module_a** (that do not start with an underscore) into main namespace. Not generally recommended.
If **__init__.py** contains: `__all__ = ['module_a', 'module_b']`	
`import my_package`	Imports my_package only, but not contents. No modules are imported. This is still not useful.
`from my_package import module_a`	As before, imports **module_a** into main namespace. Objects in **module_a** must be prefixed with **module_a**
`from my_package import *`	Imports **module_a** and **module_b**, but not **module_c** into main namespace.

If **__init__.py** contains:
`__all__ = ['module_a', 'module_b']`
`import module_a`
`import module_b`	

`import my_package`	Imports **module_a** and **module_b** into the my_package namespace. Objects in **module_a** must be prefixed with **my_package.module_a**. *Now this is useful.*
`from my_package import module_a`	Imports **module_a** into main namespace. Objects in **module_a** must be prefixed with **module_a**

Import statement	What it does
`from my_package import *`	Only imports **module_a** and **module_b** into main namespace.
`from my_package import module_c`	Imports **module_c** into the main namespace.

Documenting modules and packages

- Use docstrings
- Described in PEP 257
- Generate docs with Sphinx (optional)

In addition to comments, which are for the *maintainer* of your code, you should add docstrings, which provide documentation for the *user* of your code.

If the first statement in a module, function, or class is an unassigned string, it is assigned as the *docstring* of that object. It is stored in the special attribute *_doc_*, and so is available to code.

The docstring can use any form of literal string, but triple double quotes are preferred, for consistency.

See **PEP 257** for a detailed guide on docstring conventions.

Tools such as pydoc, and many IDEs will use the information in docstrings. In addition, the Sphinx tool will gather docstrings from an entire project and format them as a single HTML, PDF, or EPUB document.

Python style

- Code is read more often than it is written!
- Style guides enforce consistency and readability

- Indent 4 spaces (do not use tabs)
- Keep lines ⇐ 79 characters
- Imports at top of script, and on separate lines
- Surround operators with space
- Comment thoroughly to explain why and how code works when not obvious
- Use docstrings to explain how to use modules, classes, methods, and functions
- Use lower_case_with_underscores for functions, methods, and attributes
- Use UPPER_CASE_WITH_UNDERSCORES for globals
- Use StudlyCaps (mixed-case) for class names
- Use _leading_underscore for internal (non-API) attributes

Guido van Rossum, Python's BDFL (Benevolent Dictator For Life), once said that code is read much more often than it is written. This means that once code is written, it may be read by the original developer, users, subsequent developers who inherit your code. Do them a favor and make your code readable. This in turn makes your code more maintainable.

To make your code readable, it is import to write your code in a consistent manner. There are several Python style guides available, including PEP (Python Enhancement Proposal) 8, Style Guide for Python Code, and PEP 257, Docstring Conventions.

If you are part of a development team, it is a good practice to put together a style guide for the team. The team will save time not having to figure out each other's style.

Chapter 1 Exercises

Exercise 1-1 (potus.py, potus_main.py)

Create a module named to provide information from the presidents.txt file. It should provide the following function:

```
get_info(term#) -> dict  provide dictionary of info for a specified president
```

Write a script to use the module.

For the ambitious (potus_amb.py, potus_amb_main.py)

Add the following functions to the module

```
get_oldest() -> string   return the name of oldest president
get_youngest()-> string  return the name of youngest president
```

'youngest' and 'oldest' refer to age at beginning of first term and age at end of last term.

Chapter 2: Intermediate Classes

Objectives

- Defining a class and its constructor
- Creating object methods
- Adding properties to a class
- Working with class data and methods
- Leveraging inheritance for code reuse
- Implementing special methods
- Knowing when NOT to use classes

What is a class?

- Represents a *thing*
- Encapsulates functions and variables
- Creator of object *instances*
- Basic unit of object-oriented programming

A class is definition that represents a *thing*. The thing could be a file, a process, a database record, a strategy, a string, a person, or a truck.

The class describes both data, which represents one instance of the thing, and methods, which are functions that act upon the data. There can be both class data, which is shared by all instances, and instance data, which is only accessible from the instance.

TIP Classes are a very powerful tool to organize code. However, there are some circumstances in Python where classes are not needed. If you just need some functions, and they don't need to share or remember data, just put the functions in a module. If you just need some data, but you don't need functions to process it, just used a nested data structure built out of dictionaries, lists, and tuples, as needed.

Defining Classes

- Syntax

```
class ClassName(base_class,...):
    # class body – methods and data
```

- Specify base classes
- Use StudlyCaps for name

The **class** statement defines a class and assigns it to a name.

The simplest form of class definition looks like this:

```
class ClassName():
    pass
```

Normally, the contents of a class definition will be method definitions and shared data.

A class definition creates a new local namespace. All variable assignments go into this new namespace. All methods are called via the instance or the class name.

A list of base classes may be specified in parentheses after the class name.

Object Instances

- Call class name as a function
- ***self*** contains attributes
- Syntax

```
obj = ClassName(args...)
```

An object instance is an object created from a class. Each object instance has its own private attributes, which are usually created in the __init__ method.

Instance attributes

- Methods and data
- Accessed using dot notation
- Privacy by convention (_name)

An instance of a class (AKA object) normally contains methods and data. To access these attributes, use "dot notation": object.attribute.

Instance attributes are dynamic; they can be accessed directly from the object. You can create, update, and delete attributes in this way.

Attributes cannot be made private, but names that begin with an underscore are understood by convention to be for internal use only. Users of your class will not consider methods that begin with an underscore to be part of your class's API.

Example

```
class Spam():
    def eggs(self):
        pass

    def _beverage(self):   # private!
        pass

s = Spam()
s.eggs()
s.toast = 'buttered'
print(s.toast)

s._beverage()   # legal, but wrong!
```

In most cases, it is better to use properties (described later) to access data attributes.

Instance Methods

- Called from objects
- Object is implicit parameter

An instance method is a function defined in a class. When a method is called from an object, the object is passed in as the implicit first parameter, named **self** by strong convention.

Example

rabbit.py

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

class Rabbit:

    def __init__(self, size, danger):  ①
        self._size = size
        self._danger = danger
        self._victims = []

    def threaten(self):  ②
        print("I am a {} bunny with {}!".format(self._size, self._danger))


r1 = Rabbit('large', "sharp, pointy teeth")  ③
r1.threaten()  ④

r2 = Rabbit('small', 'fluffy fur')
r2.threaten()
```

① constructor, passed **self**

② instance method, passed **self**

③ pass parameters to constructor

④ instance method has access to variables via **self**

rabbit.py

```
I am a large bunny with sharp, pointy teeth!
I am a small bunny with fluffy fur!
```

Constructors

- Named __init__.py
- Implicitly called when object is created
- **self** is object it*self*

If a class defines a method named **__init__.py**, it will be automatically called when an object instance is created. This is the *constructor*.

The object being created is implicitly passed as the first parameter to **__init__.py** . This parameter is named **self** by very strong convention. Data attributes can be assigned to **self**. These attributes can then be accessed by other methods.

Example

```
class Rabbit:

    def __init__(self, size, danger):
        self._size = size
        self._danger = danger
        self._victims = []
```

TIP In C++, Java, and C#, **self** might be called **this**.

Getters and setters

- Used to access data
- AKA *accessors* and *mutators*
- Most people prefer **properties** (see next topic)

Getter and setter methods can be used to access an object's data. These are traditional in object-oriented programming.

A *getter* retrieves a data (private variable) from self. A *setter* assigns a value to a variable.

NOTE Most Python developers use *properties*, described next, instead of getters and setters.

Example

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

    def set_name(self,name):
        self._name = name

    def get_name(self):
        return self._name

k = Knight("Lancelot")
print( k.get_name() )
```

Properties

- Accessed like variables
- Invoke implicit getters and setters
- Can be read-only

While object attributes can be accessed directly, in many cases the class needs to exercise some control over the attributes.

A more elegant approach is to use properties. A property is a kind of managed attribute. Properties are accessed directly, like normal attributes (variables), but getter, setter, and deleter functions are implicitly called, so that the class can control what values are stored or retrieved from the attributes.

You can create getter, setter, and deleter properties.

To create the getter property (which must be created first), apply the **@property** decorator to a method with the name you want. It receives no parameters other than **self**.

To create the setter property, create another function with the property name (yes, there will be two function definitions with the same name). Decorate this with the property name plus ".setter". In other words, if the property is named "spam", the decorator will be "@spam.setter". The setter method will take one parameter (other than self), which is the value assigned to the property.

It is common for a setter property to raise an error if the value being assigned is invalid.

While you seldom need a deleter property, creating it is the same as for a setter property, but use "@*propertyname*.deleter".

Example

knight.py

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


class Knight():
    def __init__(self, name, title, color):
        self._name = name
        self._title = title
        self._color = color

    @property  ①
    def name(self):  ②
        return self._name

    @property
    def color(self):
        return self._color

    @color.setter  ③
    def color(self, color):
        self._color = color

    @property
    def title(self):
        return self._title


if __name__ == '__main__':
    k = Knight("Lancelot", "Sir", 'blue')

    # Bridgekeeper's question
    print('Sir {}, what is your...favorite color?'.format(k.name))  ④

    # Knight's answer
    print("red, no -- {}!".format(k.color))

    k.color = 'red'  ⑤

    print("color is now:", k.color)
```

① getter property decorator

② property implemented by name() method

③ setter property decorator

④ use property

⑤ set property

knight.py

```
Sir Lancelot, what is your...favorite color?
red, no -- blue!
color is now: red
```

Class Data

- Attached to class, not instance
- Shared by all instances

Data can be attached to the class itself, and shared among all instances. Class data can be accessed via the class name from inside or outside of the class.

Any class attribute not overwritten by an instance attribute is also available through the instance.

Example

class_data.py

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

class Rabbit:
    LOCATION = "the Cave of Caerbannog"  ①

    def __init__(self, weapon):
        self.weapon = weapon

    def display(self):
        print("This rabbit guarding {} uses {} as a weapon".
              format(self.LOCATION, self.weapon))  ②


r1 = Rabbit("a nice cup of tea")
r1.display()  ③

r1 = Rabbit("big pointy teeth")
r1.display()  ③
```

① class data

② look up class data via instance

③ instance method uses class data

class_data.py

```
This rabbit guarding the Cave of Caerbannog uses a nice cup of tea as a weapon
This rabbit guarding the Cave of Caerbannog uses big pointy teeth as a weapon
```

Class Methods

- Called from class or instance
- Use @classmethod to define
- First (implicit) parameter named "cls" by convention

If a method only needs class attributes, it can be made a class method via the @classmethod decorator. This alters the method so that it gets a copy of the class object rather than the instance object. This is true whether the method is called from the class or from an instance.

The parameter to a class method is named **cls** by strong convention.

Example

class_methods_and_data.py

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

class Rabbit:
    LOCATION = "the Cave of Caerbannog"  ①

    def __init__(self, weapon):
        self.weapon = weapon

    def display(self):
        print("This rabbit guarding {} uses {} as a weapon".
              format(self.LOCATION, self.weapon))  ②

    @classmethod  ③
    def get_location(cls):  ④
        return cls.LOCATION  ⑤


r = Rabbit("a nice cup of tea")
print(Rabbit.get_location())  ⑥
print(r.get_location())  ⑦
```

① class data (not duplicated in instances)

② instance method

③ the **@classmethod** decorator makes a function receive the class object, not the instance object

④ *get_location() is a *class* method

⑤ class methods can access class data via **cls**

⑥ call class method from class

⑦ call class method from instance

class_methods_and_data.py

```
the Cave of Caerbannog
the Cave of Caerbannog
```

Inheritance

- Specify base class in class definition
- Call base class constructor explicitly

Any language that supports classes supports *inheritance*. One or more base classes may be specified as part of the class definition. All of the previous examples in this chapter have used the default base class, object.

The base class must already be imported, if necessary. If a requested attribute is not found in the class, the search looks in the base class. This rule is applied recursively if the base class itself is derived from some other class. For instance, all classes inherit the implementation from **object**, unless a class explicitly implements it.

Classes may override methods of their base classes. (For Java and C++ programmers: all methods in Python are effectively virtual.)

To extend rather than simply replace a base class method, call the base class method directly: BaseClassName.methodname(self, arguments).

Using super()

- Follows MRO (method resolution order) to find function
- Great for single inheritance tree
- Use explicit base class names for multiple inheritance
- Syntax:

```
super().method()
```

The **super()** function can be used in a class to invoke methods in base classes. It searches the base classes and their bases, recursively, from left to right until the method is found.

The advantage of super() is that you don't have to specify the base class explicitly, so if you change the base class, it automatically does the right thing.

For classes that have a single inheritance tree, this works great. For classes that have a diamond-shaped tree, super() may not do what you expect. In this case, using the explicit base class name is best.

```
class Foo(Bar):
    def __init__(self):
        super().__init__()   # same as Bar._init__(self)
```

Example

animal.py

```
#!/usr/bin/env python
class Animal():
    count = 0  ①

    def __init__(self, species, name, sound):
        self._species = species
        self._name = name
        self._sound = sound
        Animal.count += 1

    @property
    def species(self):
        return self._species

    @classmethod
    def kill(cls):
        cls.count -= 1

    @property
    def name(self):
        return self._name

    def make_sound(self):
        print(self._sound)

    @classmethod
    def remove(cls):
        cls.count -= 1  ②

    @classmethod
    def zoo_size(cls):  ③
        return cls.count


if __name__ == "__main__":
    leo = Animal("African lion", "Leo", "Roarrrrrrr")
    garfield = Animal("cat", "Garfield", "Meowwwww")
    felix = Animal("cat", "Felix", "Meowwwww")

    print(leo.name, "is a", leo.species, "--", end=' ')
```

```
    leo.make_sound()

    print(garfield.name, "is a", garfield.species, "--", end=' ')
    garfield.make_sound()

    print(felix.name, "is a", felix.species, "--", end=' ')
    felix.make_sound()
```

① class data

② update class data from instance

③ zoo_size gets class object when called from instance or class

insect.py

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

from animal import Animal


class Insect(Animal):
    '''
        An animal with 2 sets of wings and 3 pairs of legs
    '''

    def __init__(self, species, name, sound, can_fly=True):  ①
        super().__init__(species, name, sound)  ②
        self._can_fly = can_fly

    @property
    def can_fly(self):  ③
        return self._can_fly


if __name__ == '__main__':
    mon = Insect('monarch butterfly', 'Mary', None)  ④
    scar = Insect('scarab beetle', 'Rupert', 'Bzzz', False)

    for insect in mon, scar:
        flying_status = 'can' if insect.can_fly else "can't"
        print("Hi! I am {} the {} and I {} fly!".format(  ⑤
                insect.name, insect.species, flying_status
            ),
        )
        insect.make_sound()  ⑥
        print()
```

① constructor (AKA initializer)

② call base class constructor

③ "getter" property

④ defaults to can_fly being True

⑤ .name and .species inherited from base class (Animal)

⑥ .make_sound inherited from Animal

insect.py

```
Hi! I am Mary the monarch butterfly and I can fly!
None

Hi! I am Rupert the scarab beetle and I can't fly!
Bzzz
```

Multiple Inheritance

- More than one base class
- All data and methods are inherited
- Methods resolved left-to-right, depth-first

Python classes can inherit from more than one base class. This is called "multiple inheritance".

Classes designed to be added to a base class are sometimes called "mixin classes", or just "mixins".

Methods are searched for in the first base class, then its parents, then the second base class and parents, and so forth.

Put the "extra" classes before the main base class, so any methods in those classes will override methods with the same name in the base class.

TIP To find the exact method resolution order (MRO) for a class, call the class's **mro()** method.

Example

multiple_inheritance.py

```
#!/usr/bin/env python
class AnimalBase():  ①
    def __init__(self, name):
        self._name = name


    def get_id(self):
        print(self._name)


class CanBark():  ②
    def bark(self):
        print("woof-woof")


class CanFly():  ②
    def fly(self):
        print("I'm flying")


class Dog(CanBark, AnimalBase):  ③
    pass


class Sparrow(CanFly, AnimalBase):  ③
    pass


d = Dog('Dennis')
d.get_id()  ④
d.bark()  ⑤
print()

s = Sparrow('Steve')
s.get_id()
s.fly()  ⑥
print()

print("Sparrow mro:", Sparrow.mro())
```

① create primary base class

② create additional (mixin) base class

③ inherit from primary base class plus mixin

④ all animals have id()

⑤ dogs can bark() (from mixin)

⑥ sparrows can fly() (from mixin)

multiple_inheritance.py

```
Dennis
woof-woof

Steve
I'm flying

Sparrow mro: [<class '__main__.Sparrow'>, <class '__main__.CanFly'>, <class
'__main__.AnimalBase'>, <class 'object'>]
```

Abstract base classes

- Designed for inheritance
- Abstract methods *must* be implemented
- Non-abstract methods *may* be overwritten

The **abc** module provides abstract base classes. When a method in an abstract class is designated **abstract**, it must be implemented in any derived class. If a method is not marked abstract, it may be overwritten or extended.

To create an abstract class, import ABCMeta and abstractmethod. Create the base (abstract) class normally, but assign ABCMeta to the class option **metaclass**. Then decorated any desired abstract methods with *@abstractmethod.

Now, any classes that inherit from the base class must implement any abstract methods. Non-abstract methods do not have to be implemented, but of course will be inherited.

NOTE abc also provides decorators for abstract properties and abstract class methods.

Example

abstract_base_classes.py

```
#!/usr/bin/env python
#
from abc import ABCMeta, abstractmethod

class Animal(metaclass=ABCMeta):  ①

    @abstractmethod   ②
    def speak(self):
        pass

class Dog(Animal):  ③
    def speak(self):   ④
        print("woof! woof!")

class Cat(Animal):  ③
    def speak(self):  ④
        print("Meow meow meow")

class Duck(Animal): ③
    pass  ⑤

d = Dog()
d.speak()

c = Cat()
c.speak()

try:
    d = Duck()  ⑥
    d.speak()
except TypeError as err:
    print(err)
```

① metaclasses control how classes are created; ABCMeta adds restrictions to classes that inherit from Animal

② when decorated with @abstractmethod, speak() becomes an abstract method

③ Inherit from abstract base class Animal

④ speak() **must** be implemented

⑤ Duck does not implement speak()

⑥ Duck throws a TypeError if instantiated

abstract_base_classes.py

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

Special Methods

- User-defined classes emulate standard types
- Define behavior for builtin functions
- Override operators

Python has a set of special methods that can be used to make user-defined classes emulate the behavior of builtin classes. These methods can be used to define the behavior for builtin functions such as str(), len() and repr(); they can also be used to override many Python operators, such as +, *, and ==.

These methods expect the self parameter, like all instance methods. They frequently take one or more additional methods. self. Is the object being called from the builtin function, or the left operand of a binary operator such as ==.

For instance, if your object represented a database connection, you could have str() return the hostname, port, and maybe the connection string. The default for str() is to call repr(), which returns something like <*main*.DBConn object at 0xb7828c6c>, which is not nearly so user-friendly.

TIP See http://docs.python.org/reference/datamodel.html#special-method-names for detailed documentation on the special methods.

Table 1. 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.

Method or Variables	Description
__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
__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.

Method or Variables	Description
__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 ~
__oct__(self) __hex__(self)	Implement oct() and hex() builtins
__index__(self)	Implement operator.index()
__coerce__(self, other)	Implement "mixed-mode" numeric arithmetic.

specialmethods.py

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

class Special():

    def __init__(self, value):
        self._value = str(value)  ①

    def __add__(self, other):  ②
        return self._value + other._value

    def __mul__(self, num):  ③
        return ''.join((self._value for i in range(num)))

    def __str__(self):  ④
        return self._value.upper()

    def __eq__(self, other):  ⑤
        return self._value == other._value


if __name__ == '__main__':
    s = Special('spam')
    t = Special('eggs')
    u = Special\
        ('spam')
    v = Special(5)  ⑥
    w = Special(22)

    print("s + s", s + s)  ⑦
    print("s + t", s + t)
    print("t + t", t + t)
    print("s * 10", s * 10)  ⑧
    print("t * 3", t * 3)
    print("str(s)={}  str(t)={}".format(str(s), str(t)))
    print("id(s)={} id(t)={} id(u)={}".format(id(s), id(t), id(u)))
    print("s == s", s == s)
    print("s == t", s == t)
    print("s == u", s == u)
    print("v + v", v + v)
    print("v + w", v + w)
    print("w + w", w + w)
    print("v * 10", v * 10)
```

```
    print("w * 3", w * 3)
```

① all Special instances are strings

② define what happens when a Special instance is added to another Special object

③ define what happens when a Special instance is multiplied by a value

④ define what happens when str() called on a Special instance

⑤ define equality between two Special valuess

⑥ parameter to Special() is converted to a string

⑦ add two Special instances

⑧ multiply a Special instance by an integer

specialmethods.py

```
s + s spamspam
s + t spameggs
t + t eggseggs
s * 10 spamspamspamspamspamspamspamspamspamspam
t * 3 eggseggseggs
str(s)=SPAM  str(t)=EGGS
id(s)=4509325744 id(t)=4509325800 id(u)=4509325968
s == s True
s == t False
s == u True
v + v 55
v + w 522
w + w 2222
v * 10 5555555555
w * 3 222222
```

Static Methods

- Related to class, but doesn't need instance or class object
- Use @staticmethod decorator

A static method is a utility method that is related to the class, but does not need the instance or class object. Thus, it has no automatic parameter.

One use case for static methods is to factor some kind of logic out of several methods, when the logic doesn't require any of the data in the class.

NOTE Static methods are seldom needed.

Chapter 2 Exercises

Exercise 2-1 (president.py, president_main.py)

Create a module that implements a **President** class. This class has a constructor that takes the index number of the president (1-45) and creates an object containing the associated information from the presidents.txt file.

Provide the following properties (types indicated after ->):

```
term_number -> int
first_name -> string
last_name -> string
birth_date -> date object
death_date -> date object (or None, if still alive)
birth_place -> string
birth_state -> string
term_start_date -> date object
term_end_date -> date object (or None, if still in office)
party -> string
```

Write a main script to exercise some or all of the properties. It could look something like

```
from president import President

p = President(1)   # George Washington
print("George was born at {0}, {1} on {2}".format(
    p.birth_place, p.birth_state, p.birth_date
)
```

Chapter 3: PyQt

Objectives

- Explore PyQt programming
- Understand event-driven programming
- Code a minimal PyQt application
- Use the Qt designer to create GUIs
- Wire up the generated GUI to event handlers
- Validate input
- Use predefined dialogs
- Design and use custom dialogs

What is PyQt?

- Python Bindings for Qt library
- Qt written in C++ but ported to many languages
- Complete GUI library
- Cross-platform (OS X, Windows, Linux, et al.)
- Hides platform-specific details

PyQt is a package for Python that provides binding to the generic Qt graphics programming library. Qt provides a complete graphics programming framework, and looks "native" across various platforms. In addition to graphics components, it includes database access and many other tools.

It hides the platform-specific details, so PyQt programs are portable.

Matplotlib can be integrated with PyQT for data visualizations.

NOTE The current version of PyQt is PyQt 5. There are Python packages to support both PyQt4 and PyQt5. In the EXAMPLES folder, there are subfolders for each package. The only difference in the scripts is in the PyQt imports.

Event Driven Applications

- Application starts event loop
- When event occurs, goes to handler function, then back to loop
- Terminate event ends the loop (and the app)

GUI programs are different from conventional, procedural applications. Instead of the programmer controlling the order of execution via logic, the user controls the order of execution by manipulating the GUI.

To accomplish this, starting a GUI app launches an event loop, which "listens" for user-generated events, such as key presses, mouse clicks, or mouse motion. If there is a method associated with the event (AKA "event handler") (AKA "slot" in PyQt), the method is invoked.

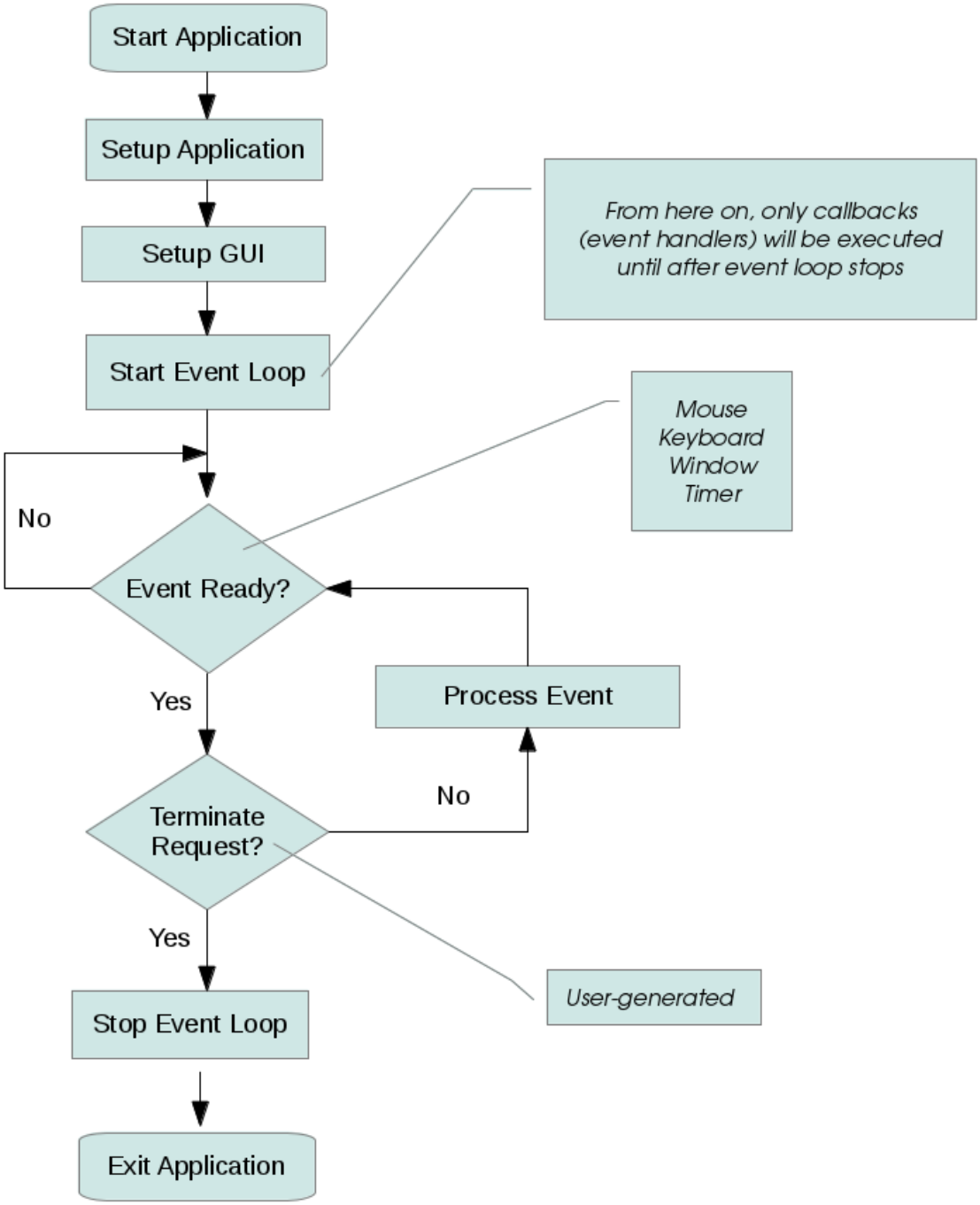
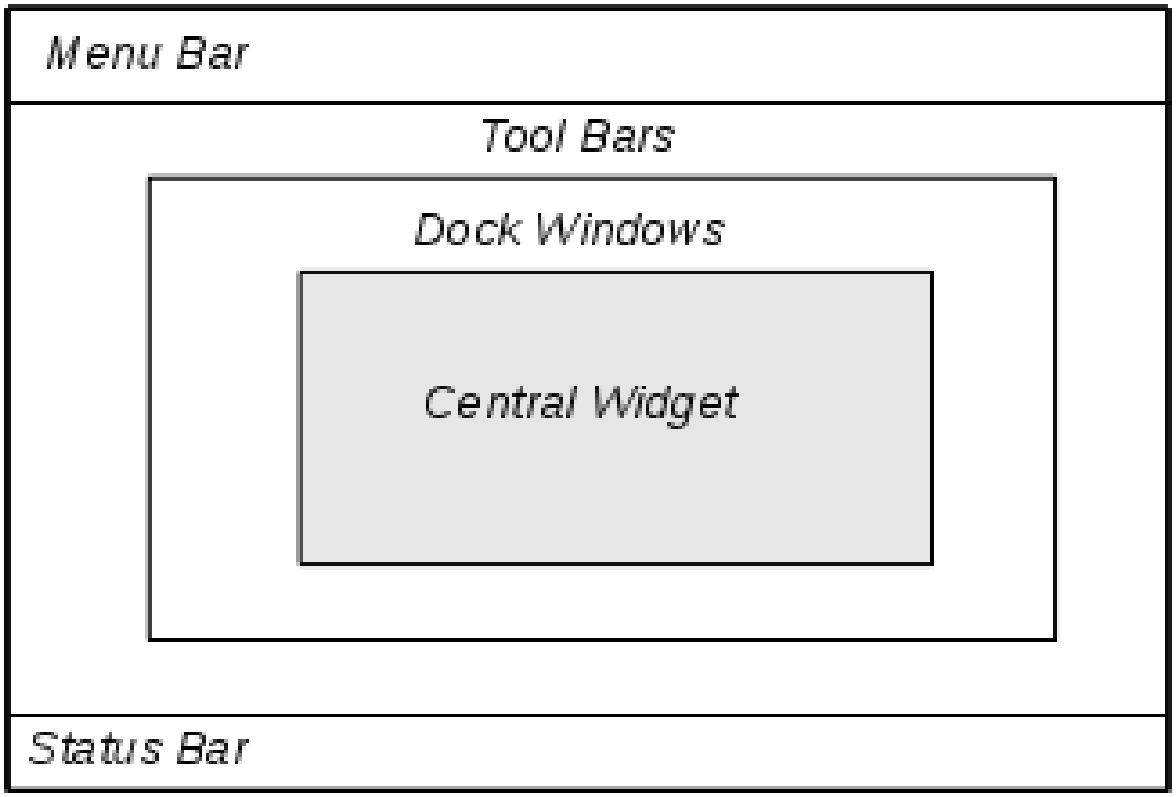


Figure 1. GUI Application Flow Chart

External Anatomy of a PyQt Application



The main window widget has several predefined areas:

- The menu bar area contains the usual File, Edit, and other standard menus.
- The tool bar areas can contain any tool bar buttons.
- The dock windows area contains any docked windows, which can be docked in any of the doc areas, or which can float freely. They have their own title bar with close, minimize and maximize buttons.
- The status bar can contain any other widgets, typically labels, but anything is fair game.
- None of the above are required, and if not present will not take up any screen space.
- The central widget is the main widget of the application. It is typically a QWidget layout object such as VBoxLayout, HBoxLayour, or GridLayout.

Internal Anatomy of a PyQt Application

- Extend (subclass) QmainWindow
- Call show() on main class

The normal (and convenient) approach is to subclass QMainWindow to create a custom main window for your application. Within this class, ***self*** is the main window of the application. You can attach widgets to and call setup methods from self.

QApplication is an object which acts as the application itself. You need to create a QApplication object and pass the command line arguments to it. To start your program call the exec_() method on your application object, after calling **show()** on the main window to make it visible.

Example

qt5/qt_hello.py

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

import sys
from PyQt5.QtWidgets import QMainWindow, QApplication, QLabel ①

class HelloWindow(QMainWindow): ②

    def __init__(self, parent = None):
        QMainWindow.__init__(self, parent)  ③
        self._label = QLabel("Hello PyQt5 World")
        self.setCentralWidget(self._label)

if __name__ == "__main__":
    app = QApplication(sys.argv)  ④
    main_window = HelloWindow()
    main_window.show()
    sys.exit(app.exec_())
```

① Standard PyQt5 imports

② Main class inherits from QMainWindow to have normal application behavior

③ Must call QMainWindow constructor

④ These 4 lines are always required. Only the name of the main window object changes.

Using designer

- GUI for building GUIs
- Builds applications, dialogs, and widgets
- Outputs generic XML that describes GUI
- pyuic5 (or pyuic4) generates Python module from XML
- Import generated module in main script

The **designer** tool makes it fast and easy to generate any kind of GUI.

To get started with designer, choose **New...** from the File menu.

Select Main Window. (You can also use designer to create dialogs and widgets). This will create a blank application window. It will already have a menu bar and a status bar. It is ready for you to drag layouts and widgets as needed.

Be sure to change the objectName property of the QMainWindow object in the Property Editor. This (with "Ui_" prefixed) will be the name of the GUI class generated by pyuic4.

To set the title of your application, which will show up on the title bar, select the QmainWindow object in the Object Inspector, then open the QWidget group of properties in the Property Editor. Enter the title in the windowTitle property.

Be sure to give your widgets meaningful names, so when you have to use them in your main program, you know which widget is which. A good approach is a short prefix that describes the kind of widget (such as "bt" for QPushButton or "cb" for QComboBox) followed by a descriptive name. Good examples are 'bt_open_file', 'cb_select_state', and 'lab_hello'. There are no standard prefixes, so use whatever makes sense to you.

You can just drag widgets onto the main window and position them, but in general you will use layouts to contain widgets.

For each of the example programs, the designer file (.ui) and the generated module are provided in addition to the source of the main script.

PyQt designer

Designer-based application workflow

- Using designer
  - Create GUI
  - Name MainWindow Hello2 (for example)
  - Save from designer as hello2.ui
- Using pyuic5 (or pyuic4)
  - generate ui_hello2 .py from hello2.ui
- Using your IDE
  - Import PyQt5 widgets
  - Subclass QMainWindow
  - Add instance of Ui_Hello2 to main window
  - Call setupUi() from Ui_Hello2
  - Instantiate main window and call show()
- Start application

Example

qt5/qt_hello2.py

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

import sys

from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_hello2 import Ui_Hello2  ①

class Hello2Main(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        ②
        self.ui = Ui_Hello2()  ③
        self.ui.setupUi(self)  ④

if __name__ == "__main__":
    app = QApplication(sys.argv)
    main = Hello2Main()
    main.show()
    sys.exit(app.exec_())
```

① import generated interface

② Set up the user interface generated from Designer.

③ Attribute name does not have to be "ui"

④ Create the widgets

Naming conventions

- Keep names consistent across applications
- Use "application name" throughout
- Pay attention to case

Using consistent names for the main window object can pay off in several ways. First, you'll be less confused. Second, you can write scripts or IDE macros to generate Python code from the designer output. Third, you can create standard templates which contain the boilerplate PyQt code to set up and run the GUI.

If application name is **Spam**:

- set QMainWindow object name to **Spam**
- set windowTitle to **Spam** (or as desired)
- Save designer file as **spam.ui**
- Redirect output of `pyuic5 spam.ui` to **ui_spam.py**
- In main program, use this import
  - `from ui_spam import Ui_Spam`

The file and object names are not required to be the same, or even similar. You can name any of these anything you like, but consistency can really help simplify code maintenance.

In **EXAMPLES/qt5** there is a file called **qt5_template.py** that you can copy, replacing AppName or appname with your app's name. This contains all of the required code for a normal Qt5 app.

NOTE

If you are using PyCharm, you can add a *live template* via **Settings(Preferences on Macs)→Editor→File and Code Templates**. Copy and paste the contents of the file **EXAMPLES/qt5/qt5_template_pycharm.py** into a live template. Be sure to set the extension to "py". The template will now ask for the app name, and insert it into the appropriate places.

Common Widgets

- QLabel, QPushButton, QLineEdit, QComboBox
- There are many more

The Qt library has many widgets; some are simple, and some are complex. Some of the most basic are Qlabel, QPushButton, QLineEdit, and QComboBox.

QLabel is a widget with some text. QPushButton is just a clickable button. QLineEdit is a one-line entry blank. QComboBox shows a list of values, and allows a new value to be entered.

QLineEdit widgets are typically paired with QLabels. Using the property editor in designer, set buddy property to be the matching QLineEdit. This allows the accelerator (specified with &letter in the label text) of the label to place focus on the paired widget.

Table 2. Common PyQt Widgets

QLabel	**Display non-editable text, image, or video**
QLineEdit	Single line input field
QPushButton	Clickable button with text or image
QRadioButton	Selectable button; only one of a radio button group can be pushed at a time
QCheckBox	Individual selectable box
QComboBox	Dropdown list of items to select; allows new entry to be added
QSpinBox	Text box for integers with up/down arrow for incrementing/decrementing
QSlider	Line with a movable handle to control a bounded value
QMenuBar	A row of QMenu widgets displayed below the title bar
QMenu	A selectable menu (can have sub-menus)
QToolBar	Panel containing buttons with text, icons or widgets
QInputDialog	Dialog with text field plus OK and Cancel buttons
QFontDialog	Font selector dialog
QColorDialog	Color selector dialog
QFileDialog	Provides several methods for selecting files and folders for opening or saving
QTab	A tabbed window. Only one QTab is visible at a time
QStacked	Stacked windows, similar to QTab
QDock	Dockable, floating, window
QStatusBar	Horizontal bar at the bottom of the main window for displaying permanent or temporary information. May contain other widgets
QList	Item-based interface for updating a list. Can be multiselectable
QScrollBar	Add scrollbars to another widget
QCalendar	Date selector

Example

qt5/qt_commonwidgets.py

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

import sys
from PyQt5.QtWidgets import QMainWindow, QApplication

from ui_commonwidgets import Ui_CommonWidgets

class CommonWidgetsMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_CommonWidgets()
        self.ui.setupUi(self)

        for k,v in (('apple',1),('banana',2),('mango',3)):  ①
            self.ui.cbFruits.insertItem(v,k,v) ①

if __name__ == "__main__":
    app = QApplication(sys.argv)
    main = CommonWidgetsMain()
    main.show()
    app.exec_()
```

① populate the combo box

Layouts

QVBoxLayout (vertical, like pancakes) QHBoxLayout (horizontal, like books on a shelf) QGridLayout (rows and columns) QFormLayout (2 columns)

Most applications use more than one widget. To easily organize widgets into rows and columns, use layouts. There are four layout types: QVBoxLayout, QHBoxLayout, QGridLayout, and QFormLayout. Drag layouts to your Deigner canvas to create the arrangement you need; of course they may be nested.

QVBoxlayout and QHBoxlayout lay out widgets vertical or horizontally. Widgets will automatically be centered and evenly spaced.

QGridLayout lays out widgets in specified rows and columns. QFormLayout is a 2-column-wide grid, for labels and input widgets.

Layouts can be resized; widgets attached to them will grow and shrink as needed (by default – all PyQt behavior can be changed in the Property editor, or programmatically).

Example

qt5/qt_layouts.py

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

import sys

from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_layouts import Ui_Layouts

class LayoutsMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_Layouts()
        self.ui.setupUi(self)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    main = LayoutsMain()
    main.show()
    sys.exit(app.exec_())
```

Selectable Buttons

- QRadioButton, QCheckBox
- Can be grouped

There are two kinds of selectable buttons. QRadioButtons are used in a group, where only one of the group can be checked. QCheckBoxes are individual, and can be checked or unchecked.

By default, radio buttons are auto-exclusive – all radio buttons that have the same parent are grouped. If you need more than one group of radio buttons to share a parent, use QButtonGroup to group them.

Use the isChecked() method to determine whether a button has been selected. Use the checked property to mark a button as checked.

Example

qt5/qt_selectables.py

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

from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_selectables import Ui_Selectables

class SelectablesMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_Selectables()
        self.ui.setupUi(self)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    main = SelectablesMain()
    main.show()
    sys.exit(app.exec_())
```

Actions and Events

- Widgets have predefined actions that can be handled
- Handler is ordinary method
- Use widget.action.connect(method)
- Pass in function object; don't call function
- Action is clicked, triggered, etc.
- Event names are predefined; override in main class

An event is something that changes in a GUI app, such as a mouse click, mouse movement, key press (or release), etc. To do something when an event occurs, you associate a function with an *action*, which represents the event. Actions are verbs that end with -ed, such as clicked, connected, triggered, etc.

To add an action to a widget, use the connect() method of the appropriate action, which is an attribute of the widget. For instance, if you have a QPushButton object pb, you can set the action with pb.clicked.connect(method).

Other events can be handled by using implementing predefined methods, such as keyPressEvent. Event handlers are passed the event object, which has more detail about the event, such as the key pressed, the mouse position, or the number of mouse clicks.

To get the widget that generated the event (AKA the sender), use self.sender() in the handler function.

If the action needs parameters, the simplest thing to do is use a lambda function as the handler (AKA slot), which can then call some other function with the desired parameters. NOTE: Actions and events are also called "signals" and "slots" in Qt.

Example

qt5/qt_events.py

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

import sys
import types

from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_events import Ui_Events

def fprint(*args):
    """ print and flush the output buffer so text
        shows up immediately
    """
    print(*args)
    sys.stdout.flush()

class EventsMain(QMainWindow):
    FRUITS = dict(A='Apple', B='Banana', C='Cherry', D='Date')

    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_Events()
        self.ui.setupUi(self)

        # set the File->Quit handler
        self.ui.actionQuit.triggered.connect(self.close) ①

        # set the Edit->Clear Name Field handler
        self.ui.actionClear_name_field.triggered.connect(self._clear_field)

        # use the same handler for all 4 buttons
        self.ui.pb_A.clicked.connect(self._mkfunc('red',self.ui.pb_A))  ②
        self.ui.pb_B.clicked.connect(self._mkfunc('blue',self.ui.pb_B))
        self.ui.pb_C.clicked.connect(self._mkfunc('yellow',self.ui.pb_C))
        self.ui.pb_D.clicked.connect(self._mkfunc('purple',self.ui.pb_D))

        self.setup_mouse_move_event_handler()

        self.ui.checkBox.toggled.connect(self._toggled)
        self.ui.checkBox.clicked.connect(self._clicked)
```

```
    def setup_mouse_move_event_handler(self):
        def mme(self, mouse_ev):
            self.ui.statusbar.showMessage("Motion: {0},{1}".format(   ③
                mouse_ev.x(), mouse_ev.y(), 0)  # 2nd param is timeout
            )
        # add method instance to label dynamically
        self.ui.label.mouseMoveEvent = types.MethodType(mme, self)

    def keyPressEvent(self, key_ev):
        """ Generated on keypresses """
        key_code = key_ev.key()   ④
        char = chr(key_code) if key_code < 128 else 'Special'  ⑤
        fprint("Key press: {0} ({1})".format(key_code, char))

    def mousePressEvent(self, mouse_ev):    ⑥
        """ generated when mouse button is pressed """
        fprint("Press:", mouse_ev.x(), mouse_ev.y())


    def mouseReleaseEvent(self, mouse_ev):
        fprint("Release:", mouse_ev.x(), mouse_ev.y())

    def _toggled(self, mouse_ev):   ⑦
        fprint("Toggle")

    def _clicked(self, mouse_ev):
        fprint("Click")

    def _checked(self, mouse_ev):
        fprint("Toggle")

    def _pushed(self):
        sender = self.sender()
        button_text = str(sender.text())
        if button_text in EventsMain.FRUITS:
            sender.setText(EventsMain.FRUITS[button_text])

    def _mkfunc(self, color, widget):  ⑧
        def pushed(stuff):
            button_text = str(widget.text())
            fprint("HI I AM BUTTON {0} and I AM {1}".format(button_text, color))
            if button_text in EventsMain.FRUITS:
                widget.setText(EventsMain.FRUITS[button_text])
```

```
        return pushed  ⑨

    def _clear_field(self):
        self.ui.leName.setText('')   ⑩

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = EventsMain()
    main.show()
    sys.exit(app.exec_())
```

① Add an event handler callback for when **Quit** is selected from the the FIle menu

② Add a handler for button A

③ Update status bar

④ Get the key that was pressed

⑤ See if it's a "normal" key

⑥ Overload mouse press event

⑦ Handler when check box is toggled

⑧ Function factory to make button click event handlers

⑨ Function factory returns….a function

⑩ Update text on the LineEntry

Signal/Slot Editor

- Event manager
  - Signal is event
  - Slot is handler
- Two ways to edit
  - Signal/Slot mode
  - Signal/Slot editor

The designer has an editor for connection signals (events) to builtin slots (handlers). You can select the widget that generates the event (emits the signal), and select which signal you want to handle. Then you can select the widget to receive the signal, and finally, select the method (on the receiving widget) to handle the event.

This is very handy for tying, for example, actionQuit.triggered to MainWindow.close()

It can't be used for custom handlers. Do that in your main script.

Editing modes

- Widgets
- Signals/Slots
- Tab Order
- Buddies

By default **designer** is in *widget editing* mode, which allows dragging new widgets onto the window and arranging them. There are three other modes.

Signal/slot editing mode lets you drag and drop to connect signals (events generated by widgets) to slots (error handling methods). You can also use the separate signal/slot editor to do this.

Tab Order editing mode lets you set the tab order of the widgets in your interface. To do this, click on the widgets in the preferred order. You can right-click on widgets for other options. Tab order is the order in which the **Tab** key will traverse the widgets.

Buddies lets you pair labels with input widgets. Accelerator keys for the labels will jump to the paired input widgets.

Menu Bar

- Add menus and sub-menus to menu bar
- Add actions to sub-menus

In the Designer, you can just start typing on the menus in the menu bar to add menu items, separators, and sub-menus. To make the menus do something, see the examples in the previous topics. Note this section of code:

```
self.ui.actionQuit.triggered.connect(lambda:self.close())
       self.ui.actionClear_name_field.triggered.connect(self._clear_field)
```

This is how to attach a callback function to a menu item. By default, the designer will name menu choices based on the text in the menu. You can name the menu items anything, however.

TIP You can also use the Signal/Slot editor in the Designer to handle menu events (signals) using builtin handlers (slots).

Status Bar

- Displays at bottom of main window

A status bar is a row at the bottom of the main window which can be used for text messages. A default status bar is automatically part of a GUI based on QmainWindow. It is named "statusbar".

To put a message on the status bar, use the showMessage() method of the status bar object. The first parameter is a string containing the message; the second parameter is the timeout in milliseconds. A timeout of 0 means display until overwritten. To clear the message, use either removeMessage(), or display an empty string.

Example

qt5/qt_statusbar.py

```
#!/usr/bin/env python
import sys
from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_statusbar import Ui_StatusBar

class StatusBarMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self._count = 0

        # Setup the user interface from Designer.
        self.ui = Ui_StatusBar()
        self.ui.setupUi(self)

        self.ui.btPushMe.clicked.connect(self._pushed)
        self._update_statusbar() ①

    def _pushed(self, ev):
        self._update_statusbar()

    def _update_statusbar(self):
        self._count += 1
        msg = "Count is " + str(self._count)
        self.ui.statusbar.showMessage(msg, 0) ②


if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = StatusBarMain()
    main.show()
    sys.exit(app.exec_())
```

① do initial status bar update

② show message, 0 means no timeout, >= 0 means timeout in seconds

Forms and validation

- Use form layout
- Validate input
- Use form data

PyQt makes it easy to create fill-in forms. Use a form layout (QFormLayout) to create a two-column form. Labels go in the left column and an entry widget goes in the right column. The entry widget can be any of a variety of widget types.

The Line Edit (QLineEdit) widget is typically used for a single-line text entry. You can add validators to this widget to accept or reject user input.

There are three kinds of validators — QRegExpValidator, QIntValidator, and QDoubleValidator. To use them, create an instance of the validator, and attach it to the Line Edit widget with the `setValidator()` method.

TIP for QRexExpValidator, you need to create a QRexExp object to pass to it.

Example

qt5/qt_validators.py

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

import sys

from PyQt5.QtWidgets import QMainWindow, QApplication
from PyQt5.QtGui import QRegExpValidator, QIntValidator
from PyQt5.QtGui import QDoubleValidator
from PyQt5.QtCore import QRegExp

from ui_validators import Ui_Validators

class ValidatorsMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        # Set up the user interface from Designer.
        self.ui = Ui_Validators()
```

```
        self.ui.setupUi(self)
        self._set_validators()

        self.ui.bt_save.clicked.connect(self._save_pushed)

    def _set_validators(self):
        # Set up the validators (could be in separate function or module)
        reg_ex = QRegExp(r"[A-Za-z0-9]{1,10}") ①
        val_alphanum = QRegExpValidator(reg_ex, self.ui.le_alphanum)  ②
        self.ui.le_alphanum.setValidator(val_alphanum) ③

        reg_ex = QRegExp(r"[a-z ]{0,30}") ①
        val_lcspace = QRegExpValidator(reg_ex, self.ui.le_lcspace)  ②
        self.ui.le_lcspace.setValidator(val_lcspace) ③

        val_nums_1_100 = QIntValidator(1, 100, self.ui.le_nums_1_100)  ④
        self.ui.le_nums_1_100.setValidator(val_nums_1_100)  ⑤

        val_float = QDoubleValidator(0.0, 20.0, 2, self.ui.le_float)  ⑥
        self.ui.le_float.setValidator(val_float)  ⑦

    def _save_pushed(self):
        alphanum = self.ui.le_alphanum.text()
        lcspace = self.ui.le_lcspace.text()
        nums = self.ui.le_nums_1_100.text()
        fl = self.ui.le_float.text()
        msg = '{}/{}/{}/{}'.format(alphanum, lcspace, nums, fl)
        self.ui.statusbar.showMessage(msg, 0) ⑧

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = ValidatorsMain()
    main.show()
    sys.exit(app.exec_())
```

① create Qt regular expression object (note use of raw string)

② create regex validator from regex object

③ attach validator to line entry field

④ create integer validator

⑤ attach validator to line entry field

⑥ create double (large float) validator

⑦ attach validator to line entry field

⑧ display valid date on status bar

Using Predefined Dialogs

- Predefined dialogs for common tasks
- Files, Messages, Colors, Fonts, Printing
- Use static methods for convenience.

PyQt defines several standard dialogs for common GUI tasks. The following chart lists them. Some of the standard dialogs can be invoked directly; however, most also provide some convenient static methods that provide more fine-grained control. These static methods return an appropriate value, typically a user selection.

Example

qt5/qt_standard_dialogs.py

```
#!/usr/bin/env python
import sys
import os

from PyQt5.QtWidgets import QMainWindow, QApplication, QFileDialog, QColorDialog,
QErrorMessage, QInputDialog
from PyQt5.QtGui import QColor

from ui_standard_dialogs import Ui_StandardDialogs

class StandardDialogsMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_StandardDialogs()
        self.ui.setupUi(self)

        self.ui.actionQuit.triggered.connect(lambda:self.close())

        # Connect up the buttons.
        self.ui.btFile.clicked.connect(self._choose_file)  ①
        self.ui.btColor.clicked.connect(self._choose_color)
        self.ui.btMessage.clicked.connect(self._show_error)
```

```
        self.ui.btInput.clicked.connect(self._get_input)
        # self.ui.BUTTON_NAME.clicked.connect(self._pushed)

    def _choose_file(self):
        full_path, _ = QFileDialog.getOpenFileName(self, 'Open file', os.getcwd())
②
        file_name = os.path.basename(full_path)
        self.ui.statusbar.showMessage("You chose: " + file_name) ③

    def _choose_color(self):
        result = QColorDialog.getColor() ④
        self.ui.statusbar.showMessage(
            "You chose #{0:02x}{1:02x}{2:02x} ({0},{1},{2})".format(
                result.red(),  ⑤
                result.green(),
                result.blue()
            )
        )


    def _show_error(self):
        em = QErrorMessage(self)  ⑥
        em.showMessage("There is a problem")
        self.ui.statusbar.showMessage('Diplaying Error')

    def _get_input(self):
        text, ok = QInputDialog.getText(self, 'Input Dialog',
            'Enter your name:')  ⑦
        if ok:
            self.ui.statusbar.showMessage("Your name is " + text)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = StandardDialogsMain()
    main.show()
    sys.exit(app.exec_())
```

① setup buttons to invoke builtin dialogs

② invoke open-file dialog (starts in current directory); returns tuple of selected file path and an empty string

③ update statusbar with chosen filename

④ invoke color selector dialog and return result

⑤ result has methods to retrieve color values

⑥ invoke error message dialog with specified message

⑦ invoke input dialog with prompt; returns entered text and boolean flag — True if user pressed OK, False if user pressed Cancel

Table 3. Standard Dialogs (with Convenience Methods)

Dialog	Description
QColorDialog.getColor	Select a color
QErrorMessage.showMessage	Display an error messages
QFileDialog.getExistingDirectory QFileDialog.getOpenFileName QFileDialog.getOpenFileNameAndFilter QFileDialog.getOpenFileNames QFileDialog.getSaveFileName QFileDialog.getSaveFileNameAndFilter	Select files or folders
QFontDialog	Select a font
QInputDialog.getText QInputDialog.getInteger QInputDialog.getDouble QInputDialog.getItem	Get input from user
QMessageBox	Display a modal message
QPageSetupDialog	Select page-related options for printer
QPrintPreviewDialog	Display print preview
QProgressDialog	Display a progress windows
QWizard	Guide user through step-by-step process

Tabs

- Use a QTab Widget
- In designer under "Containers"/a
- Each tab can have a name

A QTab Widget contains one or more tabs, each of which can contain a widget, either a single widget or some kind of container.

As usual, tabs can be created in the designer. You should give each tab a unique name, so that in your main code you can access them programmatically.

Drag a Tab Widget to your application and place it. Then you can add more tabs by right-clicking on a tab and selecting Insert Page. You can then go to the properties of the tab widget and set the properties for the currently select tab. Select other tabs and change their properties as appropriate.

The actual tabs can be on any of the 4 sides of the tab widget, and they can be left-justified, right-justified, or centered. In addition, you can modify the shape of the tabs, and of course the color and font of the labels.

Whichever tab is selected when you generate the UI file will be selected when you start your application.

NOTE The QStacked widget is similar to QTab, but can stack *any* kind of widget.

Example

qt5/qt_tabs.py

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

from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_tabs import Ui_Tabs

class TabsMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_Tabs()
        self.ui.setupUi(self)

        self.ui.actionQuit.triggered.connect(lambda:self.close())
        self.ui.actionA.triggered.connect(lambda: self._show_tab('A'))
        self.ui.actionB.triggered.connect(lambda: self._show_tab('B'))
        self.ui.actionC.triggered.connect(lambda: self._show_tab('C'))

    def _show_tab(self, which_tab):
        if which_tab == 'A':
            self.ui.tabWidget.setCurrentIndex(0)  ①
            self.ui.labA.setText('Aardvark')  ②
        elif which_tab == 'B':
            self.ui.tabWidget.setCurrentIndex(1)  ①
            self.ui.labB.setText('Bonobo')  ②
        elif which_tab == 'C':
            self.ui.tabWidget.setCurrentIndex(2)  ①
            self.ui.labC.setText('Coatimundi')  ②

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = TabsMain()
    main.show()
    sys.exit(app.exec_())
```

① choose tab programmatically

② set text on label widget on tab

Niceties

- Styling Text
- Tooltips

Fonts can be configured via the designer. Choose any widget in the Object inspector, then search for the font property group in the Property Editor. You can change the font family, the point size, and weight and slant of the text.

You can further style a widget by specifying a string containing a valid CSS (cascading style sheet). In the CSS, the selectors are the object names. Either of the following approaches can be used:

```
widget.setStyleSheet('QPushButton {color: blue}')
widget.setStyleSheet(open(cssfile).read())
```

Tooltips can be added via the designer. Search for the toolTip property and type in the desired text.

Working with Images

- Display images via QLabels
- Create a QPixMap of the image
- Assign pixmap to label

To display an image, create a Pixmap object from the graphics file. Then assign the pixmap to a QLabel. The image may be scaled or resized.

NOTE The images in the example program are scaled to fit the original label. This is why they are distorted.

Example

qt5/qt_images.py

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

from PyQt5.QtWidgets import QMainWindow, QApplication
from PyQt5.QtGui import QPixmap
from ui_images import Ui_Images

class ImagesMain(QMainWindow):

    def __init__(self):
        QMainWindow.__init__(self)

        # Set up the user interface from Designer.
        self.ui = Ui_Images()
        self.ui.setupUi(self)

        self.ui.actionQuit.triggered.connect(lambda:self.close())

        self.ui.actionPictureA.triggered.connect(
            lambda: self._show_picture('A')
        )
        self.ui.actionPictureB.triggered.connect(
            lambda: self._show_picture('B')
        )
        self.ui.actionPictureC.triggered.connect(
            lambda: self._show_picture('C')
        )

    def _show_picture(self, which_picture):
        if which_picture == 'A':
            image_file = 'apple.png'  ①
            label = self.ui.labA  ②
        elif which_picture == 'B':
            image_file = 'banana.jpg'
            label = self.ui.labB
        elif which_picture == 'C':
            image_file = 'cherry.jpg'
            label = self.ui.labC

        img = QPixmap('../../DATA/' + image_file)  ③
```

```
        label.setPixmap(img)  ④
        label.setScaledContents(True)  ⑤

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = ImagesMain()
    main.show()
    sys.exit(app.exec_())
```

① select image

② select label for image

③ create QPixMap from image

④ assign pixmap to label

⑤ scale picture

Complete Example

This is an application that lets the user load a file of words, then shows all words that match a specified regular expression

Example

qt5/qt_wordfinder.py

```
#!/usr/bin/env python
import sys
import re

from PyQt5.QtWidgets import QMainWindow, QApplication, QFileDialog
from ui_wordfinder import Ui_WordFinder

class WordFinderMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)


        # Set up the user interface from Designer.
        self.ui = Ui_WordFinder()
        self.ui.setupUi(self)

        self.ui.actionQuit.triggered.connect(lambda:self.close())

        self.ui.actionLoad.triggered.connect(self._load_file)

        self.ui.lePattern.returnPressed.connect(self._search)

        # the following might be too time-consuming for large files
        # self.ui.lePattern.textChanged.connect(self._search)

        self.ui.btSearch.clicked.connect(self._search)

    def _load_file(self):
        file_name, _ = QFileDialog.getOpenFileName(
            self, 'Open file for matching', '.'
        )
        if file_name:
```

```
            with open(file_name) as F:
                self._words = [ line.rstrip() for line in F ]

            self._numwords = len(self._words)
            self.ui.teText.clear()

            self.ui.teText.insertPlainText(
                '\n'.join(self._words))

    def _search(self):
        pattern = str(self.ui.lePattern.text())
        if pattern == '':
            pattern = '.'
        rx = re.compile(pattern)

        self.ui.teText.clear()
        self.ui.lePattern.setEnabled(False)
        # self.lePattern.setVisible(False)
        count = 0
        for word in self._words:
            if rx.search(word):
                self.ui.teText.insertPlainText(word + '\n')
                count += 1
        self.ui.lePattern.setEnabled(True)
        #self.ui.lePattern.setVisible(True)
        self.ui.statusbar.showMessage(
            "Matched {0} out of {1} words".format(count,self._numwords),
            0
        )

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = WordFinderMain()
    main.show()
    sys.exit(app.exec_())
```

Chapter 3 Exercises

Exercise 3-1 (gpresinfo.py, ui_gpresinfo.py, gpresinfo.ui)

Using the Qt designer, write a GUI application to display data from the **presidentsqlite** or **presidentmysql** module. It should have at least the following components:

- A text field (use Entry) for entering the term number
- A Search button for retrieving data
- An Exit button

A widget or widgets to display the president's information – you could use a Text widget, multiple Labels, or whatever suits your fancy.

Be creative.

For the ambitious Provide a combo box of all available presidents rather than making the user type in the name manually.

For the even more ambitious Implement a search function – let a user type text in a blank, and return a list of presidents which matches the text in either the first name or last name field. E.g., "jeff" would retrieve "Jefferson, Thomas", and "John" would retrieve "Adams, John", as well as "Kennedy, John", "Johnson, Lyndon" and so forth.

vim: set syntax=asciidoc:

Chapter 4: Regular Expressions

Objectives

- Creating regular expression objects
- Matching, searching, replacing, and splitting text
- Adding options to a pattern
- Replacing text with callbacks
- Specifying capture groups
- Using RE patterns without creating objects

Regular Expressions

- Specialized language for pattern matching
- Begun in UNIX; expanded by Perl
- Python adds some conveniences

> Regular expressions (or REs) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the re module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as Does this string match the pattern?", or Is there a match for the pattern anywhere in this string?". You can also use REs to modify a string or to split it apart in various ways.
>
> — Python Regular Expression HOWTO

Regular expressions were first popularized thirty years ago as part of Unix text processing programs such as **vi**, **sed**, and **awk**. While they were improved incrementally over the years, it was not until the advent of Perl that they substantially changed from the originals. Perl added extensions of several different kinds – shortcuts for common sequences, look-ahead and look-behind assertions, non-greedy repeat counts, and a general syntax for embedding special constructs within the regular expression itself.

Python uses Perl-style regular expressions (AKA PCREs) and adds a few extensions of its own.

RE Syntax Overview

- Regular expressions contain branches
- Branches contain atoms
- Atoms may be quantified
- Branches and atoms may be anchored

A regular expression consists of one or more branches separated by the pipe symbol. The regular expression matches any text that is matched by any of the branches.

A branch is a left-to-right sequence of atoms. Each atom consists of either a one-character match or a parenthesized group. Each atom can have a quantifier (repeat count). The default repeat count is one.

A branch can be anchored to the beginning or end of the text. Any part of a branch can be anchored to the beginning or end of a word.

TIP There is frequently only one branch.

Two good web apps for working with Python regular expressions are
https://regex101.com/#python
http://www.pythex.org/

Table 4. Regular Expression Metacharacters

Pattern	Description
.	any character
[abc]	any character in set
[^abc]	any character not in set
\w,\W	any word, non-word char
\d,\D	any digit, non-digit
\s,\S	any space, non-space char
^,$	beginning, end of string
\b	beginning or end of word
\	escape a special character
*,+,?	0 or more, 1 or more, 0 or 1
{m}	exactly m occurrences
{m,}	at least m occurrences
{m,n}	m through n occurrences
a|b	match a or b
(?aiLmsux)	Set the A, I, L, M, S, U, or X flag for the RE (see below).
(?:…)	Non-grouping version of regular parentheses.
(?P<name>…)	The substring matched by the group is accessible by name.
(?P=name)	Matches the text matched earlier by the group named name.
(?#…)	A comment; ignored.
(?=…)	Matches if … matches next, but doesn’t consume the string.
(?!…)	Matches if … doesn’t match next.
(?⇐…)	Matches if preceded by … (must be fixed length).
(?<!…)	Matches if not preceded by … (must be fixed length).

Finding matches

- Module defines static functions
- Arguments: pattern, string

There are three primary methods for finding matches.

re.search(pattern, string)

Searches s and returns the first match. Returns a match object (**SRE_Match**) on success or **None** on failure. A match object is always evaluated as **True**, and so can be used in **if** statements and **while** loops. Call the **group()** method on a match object to get the matched text.

re.finditer(pattern, string)

Provides a match object for each match found. Normally used with a **for** loop.

re.findall(pattern, string)

Finds all matches and returns a list of matched strings.

Since regular expressions generally contain many backslashes, it is usual to specify the pattern with a raw string.

Other match methods

re.match() is like match(), but searches for the pattern at beginning of s. There is an implied ^ at the beginning of the pattern.

Likewise **re.fullmatch()** only succeeds if the pattern matches the entire string. ^ and $ around the pattern are implied.

Use the search() method unless you only want to match the beginning of the string.

Example

regex_finding_matches.py

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

import re

s = """lorem ipsum M302 dolor sit amet, consectetur r99 adipiscing elit, sed do
 eiusmod tempor incididunt H476 ut labore et dolore magna Q51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A110 cupidatat non proident, sunt in H332 culpa qui
officia deserunt Y45 mollit anim id est laborum"""

pattern = r'[A-Z]\d{2,3}'  ①

if re.search(pattern, s):  ②
    print("Found pattern.")
print()

m = re.search(pattern, s)  ③
print(m)
if m:
    print("Found:", m.group(0))  ④
print()

for m in re.finditer(pattern, s):  ⑤
    print(m.group())
print()

matches = re.findall(pattern, s)  ⑥
print("matches:", matches)
```

① store pattern in raw string

② search returns True on match

③ search actually returns match object

④ group() returns text that was matched

⑤ iterate over all matches in string:

⑥ return list of all matches

regex_finding_matches.py

```
Found pattern.

<re.Match object; span=(12, 16), match='M302'>
Found: M302

M302
H476
Q51
U901
A110
H332
Y45

matches: ['M302', 'H476', 'Q51', 'U901', 'A110', 'H332', 'Y45']
```

RE Objects

- **re** object contains a compiled regular expression
- Call methods on the object, with strings as parameters.

An **re** object is created by calling the compile() function, from the **re** module, with a pattern string. Once created, the object can be used for searching (matching), replacing, and splitting any string. The re.compile() function has an optional argument for flags which enable special features or fine-tune the match.

TIP It is generally a good practice to create your re objects in a location near the top of your script, and then use them as necessary

Example

regex_objects.py

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

import re

s = """lorem ipsum M302 dolor sit amet, consectetur r99 adipiscing elit, sed do
 eiusmod tempor incididunt H476 ut labore et dolore magna Q51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo Z883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A110 cupidatat non proident, sunt in H332 culpa qui
officia deserunt Y45 mollit anim id est laborum"""

rx_code = re.compile(r'[A-Z]\d{2,3}', re.I)  ①

if rx_code.search(s):  ②
    print("Found pattern.")
print()

m = rx_code.search(s)
if m:
    print("Found:", m.group())
print()

for m in rx_code.finditer(s):
    print(m.group())
print()

matches = rx_code.findall(s)
print("matches:", matches)
```

① Create an re (regular expression) object

② Call search() method from the object

regex_objects.py

```
Found pattern.

Found: M302

M302
r99
H476
Q51
Z883
U901
A110
H332
Y45

matches: ['M302', 'r99', 'H476', 'Q51', 'Z883', 'U901', 'A110', 'H332', 'Y45']
```

Compilation Flags

- Control match
- Add features

When compiling a pattern, you can specify various flags to control how the match occurs. The flags are aliases for numeric values, and can be combined by ORing them together Each flag has a short for and a long form.

re.I, re.IGNORECASE

Perform case-insensitive matching; character class and literal strings will match letters by ignoring case. For example, [A-Z] will match lowercase letters, too, and Spam will match "Spam", "spam", or "spAM". This lower-casing doesn't take the current locale into account; it will if you also set the LOCALE flag.

re.L, re.LOCALE

Make \w, \W, \b, and \B, dependent on the current locale.

Locales are a feature of the C library intended to help in writing programs that take account of language differences. For example, if you're processing French text, you'd want to be able to write \w+ to match words, but \w only matches the character class [A-Za-z]; it won't match "é" or "ç". If your system is configured properly and a French locale is selected, certain C functions will tell the program that "é" should also be considered a letter. Setting the LOCALE flag enables \w+ to match French words as you'd expect.

re.M, re.MULTILINE

Usually ^ matches only at the beginning of the string, and $ matches only at the end of the string and immediately before the newline (if any) at the end of the string. When this flag is specified, ^ matches at the beginning of the string and at the beginning of each line within the string, immediately following each newline. Similarly, the $ metacharacter matches either at the end of the string and at the end of each line (immediately preceding each newline).

re.S, re.DOTALL

Makes the "." special character match any character at all, including a newline; without this flag, "." will match anything except a newline.

re.X, re.VERBOSE

This flag allows you to write regular expressions that are more readable by granting you more flexibility in how you can format them. When this flag has been specified, whitespace within the RE string is ignored, except when the whitespace is in a character class or preceded by an unescaped backslash; this lets you organize and indent the RE more clearly. It also enables you to put comments within a RE that will be ignored by the engine; comments are marked by a "#" that's neither in a character class or preceded by an unescaped backslash. Use a triple-quoted string for your pattern to make best advantage of this flag.

Example

regex_flags.py

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

import re

s = """lorem ipsum M302 dolor sit amet, consectetur r99 adipiscing elit, sed do
 eiusmod tempor incididunt H476 ut labore et dolore magna Q51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo Z883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A110 cupidatat non proident, sunt in H332 culpa qui
officia deserunt Y45 mollit anim id est laborum"""

pattern = r'[A-Z]\d{2,3}'

if re.search(pattern, s, re.IGNORECASE):  ①
    print("Found pattern.")
print()

m = re.search(pattern, s, re.I)  ②
if m:
    print("Found:", m.group())
print()

for m in re.finditer(pattern, s, re.I):
    print(m.group())
print()

matches = re.findall(pattern, s, re.I)
print("matches:", matches)
```

① make search case-insensitive

② short version of flag

regex_flags.py

```
Found pattern.

Found: M302

M302
r99
H476
Q51
Z883
U901
A110
H332
Y45

matches: ['M302', 'r99', 'H476', 'Q51', 'Z883', 'U901', 'A110', 'H332', 'Y45']
```

Groups

- Marked with parentheses
- Capture whatever matched pattern within
- Access with match.group()

Frequently you need to obtain more information than just whether the RE matched or not. Regular expressions are often used to dissect strings by writing a RE divided into several subgroups which match different components of interest. For example, an RFC-822 header line is divided into a header name and a value, separated by a ":". This can be handled by writing a regular expression which matches an entire header line, and has one group which matches the header name, and another group which matches the header's value.

Groups are marked with parentheses, and 'capture' whatever matched the pattern inside the parentheses.

re.findall() returns a list of tuples, where each tuple contains the match for each group.

To access groups in more detail, use **finditer()** and call the **group()** method on each match object. The default group is 0, which is always the entire match. It can be retrieved with either ***match*.group(0)**, or just ***match*.group()**. ***match*.group(1)** returns text matched by the first set of parentheses, ***match*.group(2)** returns the text from the second set, etc.

In the same vein, ***match*.start()** or ***match*.start(0)** return the beginning 0-based offset of the entire match; ***match*.start(1)** returns the beginning offset of group 1, and so forth. The same is true for ***match*.end()** and ***match*.end(n)**.

***match*.span()** returns the the start and end offsets for the entire match. ***match*.span(1)** returns start and end offsets for group 1, and so forth.

Example

regex_group.py

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

import re

s = """lorem ipsum M302 dolor sit amet, consectetur r99 adipiscing elit, sed do
 eiusmod tempor incididunt H476 ut labore et dolore magna Q51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo Z883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A110 cupidatat non proident, sunt in H332 culpa qui
officia deserunt Y45 mollit anim id est laborum"""

pattern = r'([A-Z])(\d{2,3})'  ①

for m in re.finditer(pattern, s):
    print(m.group(0), m.group(1), m.group(2))  ②
print()

matches = re.findall(pattern, s)  ③
print("matches:", matches)
```

① parens delimit groups

② group 1 is first group, etc. (group 0 is entire match)

③ findall() returns list of tuples containing groups

regex_group.py

```
M302 M 302
H476 H 476
Q51 Q 51
Z883 Z 883
U901 U 901
A110 A 110
H332 H 332
Y45 Y 45

matches: [('M', '302'), ('H', '476'), ('Q', '51'), ('Z', '883'), ('U', '901'), ('A',
'110'), ('H', '332'), ('Y', '45')]
```

Special Groups

- Non-capture groups are used just for grouping
- Named groups allow retrieval of sub-expressions by name rather than number
- Look-ahead and look-behind match, but do not capture

There are two variations on RE groups that are useful. If the first character inside the group is a question mark, then the parentheses contain some sort of extended pattern, designated by the next character after the question mark. The most basic is (?:pattern), which groups but does not capture.

A welcome addition in Python is the concept of named groups. Instead of remembering that the month is the 3rd group and the year is the 4th group, you can use the syntax (?P<name>pattern). You can then call match.group("name") to fetch the text match by that sub-expression; alternatively, you can call match.groupdict(), which returns a dictionary where the keys are the pattern names, and the values are the text matched by each pattern.

Another advanced concept is an assertion, either lookahead or lookbehind. A lookahead assertion uses the syntax (?=pattern). The string being matched must match the lookahead, but does not become part of the overall match.

For instance, "\d(?st|nd|rd|th)(?=street)" matches "1st", "2nd", etc., but only where they are followed by "street".

Example

regex_special.py

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

import re

s = """lorem ipsum M302 dolor sit amet, consectetur r99 adipiscing elit, sed do
 eiusmod tempor incididunt H476 ut labore et dolore magna Q51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo Z883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A110 cupidatat non proident, sunt in H332 culpa qui
officia deserunt Y45 mollit anim id est laborum"""

pattern = r'(?P<letter>[A-Z])(?P<number>\d{2,3})'  ①

for m in re.finditer(pattern, s):
    print(m.group('letter'), m.group('number'))  ②
```

① Use (?<NAME>...) to name groups

② Use m.group(NAME) to retrieve text

regex_special.py

```
M 302
H 476
Q 51
Z 883
U 901
A 110
H 332
Y 45
```

Replacing text

- Use RE.sub(replacement,string[,count])
- RE.subn() returns tuple with string and count

To find and replace text using a regular expression, use the sub() method. It takes the replacement text and the string to search as arguments, and returns the modified string.

The third, optional argument is the maximum number of replacements to make.

Be sure to put the arguments in the proper order!

Example

regex_sub.py

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

import re

s = """lorem ipsum M302 dolor sit amet, consectetur r99 adipiscing elit, sed do
 eiusmod tempor incididunt H476 ut labore et dolore magna Q51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo Z883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A110 cupidatat non proident, sunt in H332 culpa qui
officia deserunt Y45 mollit anim id est laborum"""

rx_code = re.compile(r'(?P<letter>[A-Z])(?P<number>\d{2,3})', re.I)

s2 = rx_code.sub("[REDACTED]", s) ①
print(s2)
print()

s3, count = rx_code.subn("___", s) ②
print("Made {} replacements".format(count))
print(s3)
```

① replace pattern with string

② subn returns tuple with result string and replacement count

regex_sub.py

```
lorem ipsum [REDACTED] dolor sit amet, consectetur [REDACTED] adipiscing elit, sed
do
 eiusmod tempor incididunt [REDACTED] ut labore et dolore magna [REDACTED] aliqua.
Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo [REDACTED]  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore [REDACTED] eu fugiat nulla pariatur.
Excepteur sint occaecat [REDACTED] cupidatat non proident, sunt in [REDACTED] culpa
qui
officia deserunt [REDACTED] mollit anim id est laborum

Made 9 replacements
lorem ipsum ___ dolor sit amet, consectetur ___ adipiscing elit, sed do
 eiusmod tempor incididunt ___ ut labore et dolore magna ___ aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo ___  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore ___ eu fugiat nulla pariatur.
Excepteur sint occaecat ___ cupidatat non proident, sunt in ___ culpa qui
officia deserunt ___ mollit anim id est laborum
```

Replacing with a callback

- Replacement can be function
- Function expects match object, returns replacement text
- Use normally defined function or a lambda

In addition using a string as the replacement, you can specify a function. This function will be called once for each match, and passed in the match object. Whatever string the function returns will be used as the replacement text. This lets you have complete control over the replacement.

Using a callback makes it simple to do things such as:

- preserving case in a replacement
- adding text around the replacement
- looking up the original text in a dictionary or database

Example

regex_sub_callback.py

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

import re

s = """lorem ipsum M302 dolor sit amet, consectetur r99 adipiscing elit, sed do
 eiusmod tempor incididunt H476 ut labore et dolore magna Q51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo Z883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A110 cupidatat non proident, sunt in H332 culpa qui
officia deserunt Y45 mollit anim id est laborum"""

#  my $rx_code = qr/(?P<letter>[A-Z])(?P<number>\d{2,3})/i;
#  if ($foo ~= /$rx_code/) { }

rx_code = re.compile(r'(?P<letter>[A-Z])(?P<number>\d{2,3})', re.I)


def update_code(m):  ①
    letter = m.group('letter').upper()
    number = int(m.group('number'))
    return '{}{:04d}'.format(letter, number)  ②


s2 = rx_code.sub(update_code, s)  ③
print(s2)
```

① callback function is passed each match object

② function returns replacement text

③ sub takes callback function instead of replacement text

regex_sub_callback.py

```
lorem ipsum M0302 dolor sit amet, consectetur R0099 adipiscing elit, sed do
 eiusmod tempor incididunt H0476 ut labore et dolore magna Q0051 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo Z0883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U0901 eu fugiat nulla pariatur.
Excepteur sint occaecat A0110 cupidatat non proident, sunt in H0332 culpa qui
officia deserunt Y0045 mollit anim id est laborum
```

Splitting a string

- Syntax: re.split(string[,max])

The re.split() method splits a string into pieces, returning the pieces as a list. The optional max argument limits the numbers of pieces.

Example

regex_split.py

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

import re

rx_wordsep = re.compile(r"[^a-z]+")  ①

s1 = '''There are 10 kinds of people in a Binary world, I hear" -- Geek talk'''

words = rx_wordsep.split(s1)  ②
print(words)
```

① When splitting, pattern matches what you **don't** want

② Retrieve text *separated* by your pattern

regex_split.py

```
['', 'here', 'are', 'kinds', 'of', 'people', 'in', 'a', 'inary', 'world', 'hear',
'eek', 'talk']
```

Chapter 4 Exercises

Exercise 4-1 (pyfind.py)

Write a script which takes two or more arguments. The first argument is the pattern to search for; the remaining arguments are files to search. For each file, print out all lines which match the pattern.1

Exercise 4-2 (mark_big_words.py)

Copy parrot.txt to bigwords.txt adding asterisks around all words that are 8 or more characters long.

HINT: Use the \b anchor to indicate beginning and end of a word.

Exercise 4-3 (print_numbers.py)

Write a script to print out all lines in custinfo.dat which contain phone numbers.

Exercise 4-4 (word_freq.py)

Write a script that will read a text file and print out a list of all the words in the file, normalized to lower case, and with the number of times that word occurred in the file. Use the regular expression [^\w']+ for splitting each line into words.

Test with any of the text files in the DATA folder.

Chapter 5: 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
- Can have one wild card
- Frequently used with list of tuples

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

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


people = [  ①
    ('Melinda', 'Gates', 'Gates Foundation'),
    ('Steve', 'Jobs', 'Apple'),
    ('Larry', 'Wall', 'Perl'),
    ('Paul', 'Allen', 'Microsoft'),
    ('Larry', 'Ellison', 'Oracle'),
    ('Bill', 'Gates', 'Microsoft'),
    ('Mark', 'Zuckerberg', 'Facebook'),
    ('Sergey', 'Brin', 'Google'),
    ('Larry', 'Page', 'Google'),
    ('Linus', 'Torvalds', 'Linux'),
]

for first_name, last_name, org in people:  ②
    print("{} {}".format(first_name, last_name))
```

① A list of 3-element tuples

② The for loop unpacks each tuple into the three variables.

unpacking_people.py

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

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

people = [ ①
    ('Joe', 'Schmoe', 'Burbank', 'CA'),
    ('Mary', 'Rattburger', 'Madison', 'WI'),
    ('Jose', 'Ramirez', 'Ames', 'IA'),
]

def person_record(first_name, last_name, city, state): ②
    print("{} {} lives in {}, {}".format(first_name, last_name, city, state))

for person in people:  ③
    person_record(*person)  ④
```

① list of 4-element tuples

② function that takes 4 parameters

③ person is a tuple (one element of people list)

④ ***person** unpacks the tuple into four individual parameters

unpacking_function_args.py

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

Example

shoe_sizes.py

```
#!/usr/bin/env python
#
BARLEYCORN = 1 / 3.0
CM_TO_INCH = 2.54
MENS_START_SIZE = 12
WOMENS_START_SIZE = 10.5

FMT = '{:6.1f} {:8.2f} {:8.2f}'
HEADFMT = '{:>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((HEADFMT.format(*HEADINGS)))  ①
        for size in SIZE_RANGE:
            inches, cm = get_length(size, flag)
            print(FMT.format(size, inches, cm))

        print()

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

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

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

① format expects individual arguments for each placeholder; the asterisk unpacks HEADINGS into individual strings

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

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

"""Basic sorting example"""

fruit = ["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(fruit)  ①

print(sorted_fruit)
```

① sorted() returns a list

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

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

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):  ①
    return item.lower()  ②

fs1 = sorted(fruit, key=ignore_case)  ③
print("Ignoring case:")
print(" ".join(fs1), end="\n\n")

def by_length_then_name(item):
    return (len(item), item.lower())  ④

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)  ⑤
print("Numbers sorted numerically:")
for n in n1:
    print(n, end=' ')
print("\n")

n2 = sorted(nums, key=str)  ⑥
print("Numbers sorted as strings:")
for n in n2:
    print(n, end=' ')
print()
```

① Parameter is *one* element of iterable to be sorted

② Return value to sort on

③ Specify function with named parameter **key**

④ Key functions can return tuple of values to compare, in order

⑤ Numbers sort numerically by default

⑥ Sort numbers as strings

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

```
#!/usr/bin/env python
"""Sort titles, ignoring leading articles"""
import re

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",
]

rx_article = re.compile(r'^(the|a|an)\s+', re.I)  ①

def strip_articles(title):  ②
    stripped_title = rx_article.sub('', title.lower())  ③
    return stripped_title

for book in sorted(books, key=strip_articles):  ④
    print(book)
```

① compile regex to match leading articles

② create function which takes element to compare and returns comparison key

③ strip off article and convert title to lower case

④ sort using custom function

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_examples.py

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

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

sfruits = sorted(fruits, key=lambda e: e.lower())  ①

print(" ".join(sfruits))
```

① The lambda function takes one fruit and returns it in lower case

lambda_examples.py

```
Apple apricot guava KIWI LEMON Mango 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

```
#!/usr/bin/env 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]  ①

afruits = [fruit for fruit in fruits if fruit.startswith('a')]  ②

doubles = [v * 2 for v in values]  ③

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

① Copy each fruit to upper case

② Select each fruit if it starts with 'a'

③ Copy each number, doubling it

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

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


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

d = {a.lower(): len(a) for a in animals}  ①

print(d, '\n')

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

d = {w:{c:w.count(c) for c in sorted(w)} for w in words} ②

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

① Create a dictionary with key/value pairs derived from an iterable

② Use a nested dictionary comprehension to create a dictionary mapping words to dictionaries which map letters to their counts (could be useful for anagrams)

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

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

import re

with open("../DATA/mary.txt") as mary_in:
    s = {w.lower()  for ln in mary_in  for w in re.split(r'\W+', ln) if w} ①
print(s)
```

① Get unique words from file. Only one line is in memory at a time. Skip "empty" words.

set_comprehension.py

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

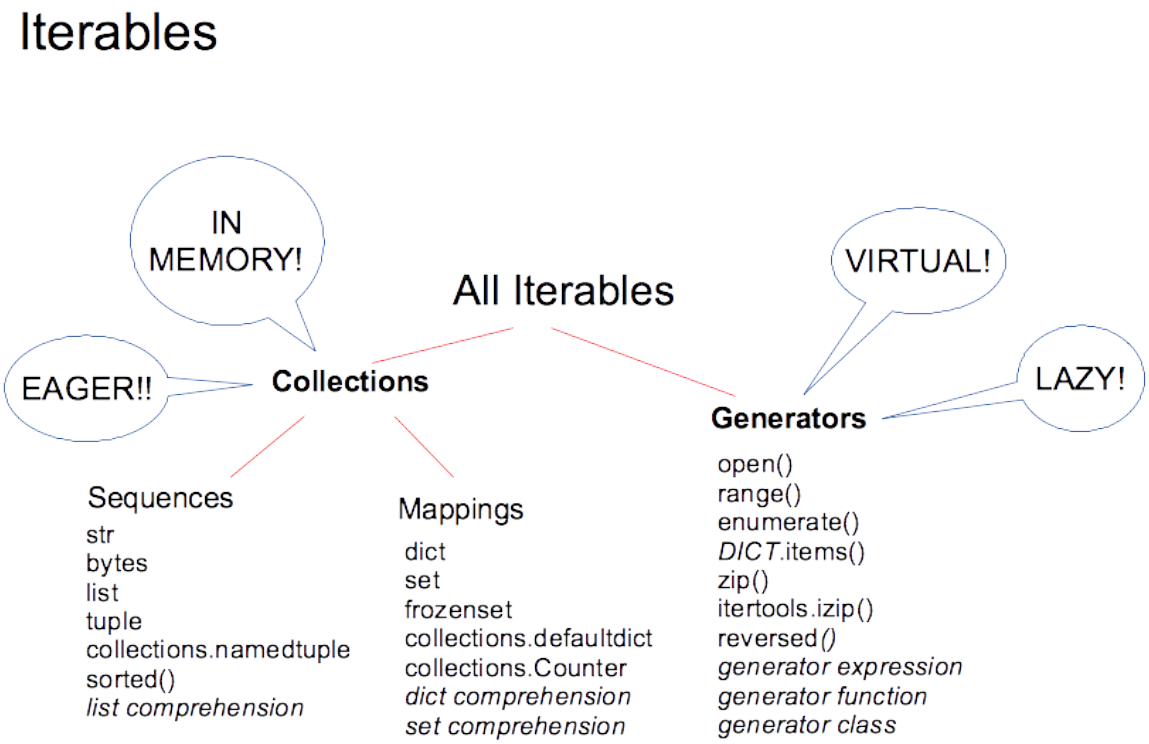
Iterables

- 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

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

# 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)])  ①

# only one square is in memory at a time with generator expression
s2 = sum(x * x for x in range(10))  ②
print(s1, s2)
print()

page = open("../DATA/mary.txt")
m = max(len(line) for line in page)  ③
page.close()
print(m)
```

① using list comprehension, entire list is stored in memory

② with generator expression, only one square is in memory at a time

③ only one line in memory at a time. max() iterates over generated values

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

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

def next_prime(limit):
    flags = set()  ①

    for i in range(2, limit):
        if i in flags:
            continue
        for j in range(2 * i, limit + 1, i):
            flags.add(j)  ②
        yield i  ③


np = next_prime(200)  ④
for prime in np:  ⑤
    print(prime, end=' ')
```

① initialize empty set (to be used for "is-prime" flags

② add non-prime elements to set

③ execution stops here until next value is requested by for-in loop

④ next_prime() returns a generator object

⑤ iterate over **yielded** primes

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

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

def trimmed(file_name):
    with open(file_name) as file_in:
        for line in file_in:
            if line.endswith('\n'):
                line = line.rstrip('\n\r')
            yield line  ①


for trimmed_line in trimmed('../DATA/mary.txt'):  ②
    print(trimmed_line)
```

① 'yield' causes this function to return a generator object

② looping over the a generator object returned by trimmed()

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

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

from datetime import date

color = 'blue'
animal = 'iguana'

print('{} {}'.format(color, animal))  ①

fahr = 98.6839832
print('{:.1f}'.format(fahr))  ②

value = 12345
print('{0:d} {0:04x} {0:08o} {0:016b}'.format(value))  ③

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

for letter, number in sorted(data.items()):
    print("{} {:4d}".format(letter, number))  ④
```

① {} placeholders are autonumbered, starting at 0; this corresponds to the parameters to format()

② Formatting directives start with ':'; .1f means format floating point with one decimal place

③ {} placeholders can be manually numbered to reuse parameters

④ :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

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

import sys

if sys.version_info.major == 3 and sys.version_info.minor >= 6:

    name = "Tim"
    count = 5
    avg = 3.456
    info = 2093
    result = 38293892

    print(f"Name is [{name:<10s}]")  ①
    print(f"Name is [{name:>10s}]")  ②
    print(f"count is {count:03d} avg is {avg:.2f}")  ③

    print(f"info is {info} {info:d} {info:o} {info:x}")  ④

    print(f"${result:,d}")  ⑤

    city = 'Orlando'
    temp = 85

    print(f"It is {temp} in {city}")  ⑥

else:
    print("Sorry -- f-strings are only supported by Python 3.6+")
```

① < means left justify (default for non-numbers), 10 is field width, s formats a string

② > means right justify

③ .2f means round a float to 2 decimal points

④ d is decimal, o is octal, x is hex

⑤ , means add commas to numeric value

⑥ parameters can be selected by name instead of position

f_strings.py

```
Name is [Tim       ]
Name is [       Tim]
count is 005 avg is 3.46
info is 2093 2093 4055 82d
$38,293,892
It is 85 in Orlando
```

Chapter 5 Exercises

Exercise 5-1 (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 5-2 (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 5-3 (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.

Optional Material

The following sections are optional, and will be covered as time permits.

Unresolved directive in py3bnpinterm.asc - include::/Users/jstrick/curr/courses/python/chapters3/scripting_for_system_administration_3.asc[] :chapternum: 1 = Chapter 5: Working with Files

Objectives

- Reading a text file line-by-line
- Reading an entire text files
- Reading all lines of a text file into an array
- Writing to a text file

Text file I/O

- Create a file object with open
- Specify modes: read/write, text/binary
- Read or write from file object
- Close file object (or use **with** block)

Python provides a file object that is created by the built-in open() function. From this file object you can read or write data in several different ways. When opening a file, you specify the file name and the mode, which says whether you want to read, write, or append to the file, and whether you want text or binary (raw) processing.

NOTE This chapter is about working with generic files. For files in standard formats, such as XML, CSV, YAML, JSON, and many others, Python has format-specific modules to read them.

Opening a text file

- Specify the file name and the mode
- Returns a file object
- Mode can be read or write
- Specify "b" for binary (raw) mode
- Omit mode for reading

Open a text file with the open() command. Arguments are the file name, which may be specified as a relative or absolute path, and the mode. The mode consists of "r" for read, "w" for write, or "a" for append. To open a file in binary mode, add "b" to the mode, as in "rb", "wb", or "ab".

If you omit the mode, "r" is the default.

Example

```
ty = open("tyger.txt","r")  open for reading in text mode
ty = open("tyger.txt")      open for reading in text mode (default mode)
junk = open("junk.dat","rb")    open for reading in raw mode
stf = open("stuff.txt","w") open for writing in text mode
stf = open("stuff.txt","x") open for writing in text mode, fail if file exists
moju = open("morejunk.dat","wb")    open for writing in raw mode
config = open("spam.cfg","a")   open for append in text mode
```

TIP The **fileinput** module in the standard library makes it easy to loop over each line in all files specified on the command line, or STDIN if no files are specified. This avoids having to open and close each file.

The *with* block

- Provides "execution context"
- Automagically closes file object
- Not specific to file objects

Because it is easy to forget to close a file object, you can use a **with** block to open your file. This will automatically close the file object when the block is finished. The syntax is

```
with open(filename, mode) as fileobject:
    # process fileobject
```

Reading a text file

- Iterate through file with for/in

```
    for line in file_in
```

- Use methods of the file object

```
    file_in.readlines()   read all lines from file_in
    file_in.read()        read all of file_in
    file_in.read(n)       read n characters from file in text mode; n bytes
from file_in in binary mode
    file_in.readline()    read next line from file_in
```

The easiest way to read a file is by looping through the file object with a for/in loop. This is possible because the file object is an iterator, which means the object knows how to provide a sequence of values.

You can also read a text file one line or multiple lines at a time. **readline()** reads the next available line; **readlines()** reads all lines into a list.

read() will read the entire file; **read(n)** will read n bytes from the file (n *characters* if in text mode).

readline() will read the next line from the file.

Example

read_tyger.py

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

with open("../DATA/tyger.txt", "r") as tyger_in:  ①
    for line in tyger_in:  ②
        print(line, end='')  ③
```

① **tyger_in** is return value of **open(...)**

② **tyger_in** is a *generator*, returning one line at a time

③ the line already has a newline, so **print()** does not need one

read_tyger.py

```
           The Tyger

Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand or eye
Could frame thy fearful symmetry?

In what distant deeps or skies
Burnt the fire of thine eyes?
On what wings dare he aspire?
What the hand dare seize the fire?

And what shoulder, & what art,
Could twist the sinews of thy heart?
And when thy heart began to beat,
What dread hand? & what dread feet?

What the hammer? what the chain?
In what furnace was thy brain?
What the anvil? what dread grasp
Dare its deadly terrors clasp?

When the stars threw down their spears
And water'd heaven with their tears,
Did he smile his work to see?
Did he who made the Lamb make thee?

Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand or eye
Dare frame thy fearful symmetry?

                    by William Blake
```

Example

reading_files.py

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

print("** About Spam **")
with open("../DATA/spam.txt") as spam_in:
    for line in spam_in:
        print(line.rstrip('\r\n')) ①

with open("../DATA/eggs.txt") as eggs_in:
    eggs = eggs_in.readlines()  ②

print("\n\n** About Eggs **")
print(eggs[0].rstrip())  ③
print(eggs[2].rstrip())
```

① rstrip('\n\r') removes \r or \n from end of string

② readlines() reads all lines into an array

③ [:-1] is another way to skip the newline (but .rstrip() does not remove spaces or tabs if they are significant)

reading_files.py

```
** About Spam **
SPAM may be famous now, but it wasn't always that way. Fact is, SPAM hails from some
rather humble beginnings.

Flash back to Austin, Minnesota, in 1937.

You're right. There isn't much here, except for an ambitious company called Hormel.

These good folks are about to hit upon an amazing little recipe: a spicy ham
packaged in a handy dandy 12-ounce can.

J. C. Hormel, then president, adds the crowning ingredient: He holds a contest to
give the product a name as distinctive as its taste.

SPAM soars. In fact, in that very first year of production, it grabs 18 percent of
the market.

Over 65 years years later, more than 6 billion cans of SPAM have been sold.


** About Eggs **
You can scramble, fry, poach and bake eggs or cook them in their shells.
Eggs are also the main ingredient in some dishes that came to the U.S.  from other
countries, such as a frittata, egg foo yung, quiche or souffle.
```

Writing to a text file

- Use write() or writelines()
- Add \n manually

To write to a text file, use the write() function to write a single string; or writelines() to write a list of strings.

writelines() will not add newline characters, so make sure the items in your list already have them.

Example

write_file.py

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

states = (
    'Virginia',
    'North Carolina',
    'Washington',
    'New York',
    'Florida',
    'Ohio',
)

with open("states.txt", "w") as states_out: ①
    for state in states:
        states_out.write(state + "\n")  ②
```

① "w" opens for writing, "a" for append

② write() does not add \n automatically

write_file.py

cat states.txt (or type states.txt under windows)

```
Virginia
North Carolina
Washington
New York
Florida
Ohio
```

"writelines" should have been called "writestrings"

Table 5. File Methods

Function	Description
f.close()	close file f
f.flush()	write out buffered data to file f
s = f.read(n) s = f.read()	read size bytes from file f into string s; if n is ⇐ 0, or omitted, reads entire file
s = f.readline() s = f.readline(n)	read one line from file f into string s. If n is specified, read no more than n characters
m = f.readlines()	read all lines from file f into list m
f.seek(n) f.seek(n,w)	position file f at offset n for next read or write; if argument w (whence) is omitted or 0, offset is from beginning; if 1, from current file position, if 2, from end of file
f.tell()	return current offset from beginning of file
f.write(s)	write string s to file f
f.writelines(m)	write list of strings m to file f; does not add line terminators

Chapter 5 Exercises

Exercise 5-1 (line_no.py)

Write a program to display each line of a file preceded by the line number. Allow your program to process one or more files specified on the command line. Be sure to reset the line number for each file.

TIP Use enumerate().

Test with the following commands:

```
python line_no.py DATA/tyger.txt
python line_no.py DATA/parrot.txt DATA/tyger.txt
```

Test with other files, as desired

Exercise 5-2 (alt_lines.py)

Write a program to create two files, a.txt and b.txt from the file alt.txt. Lines that start with 'a' go in a.txt; the other lines (which all start with 'b') go in b.txt. Compare the original to the two new files.

Exercise 5-3 (count_alice.py, count_words.py)

A. Write a program to count how many lines of alice.txt contain the word "Alice". (There should be 392).

TIP Use the **in** operator to test whether a line contains the word "Alice"

B. Modify count_alice.py to take the first command line parameter as a word to find, and the remaining parameters as filenames. For each file, print out the file name and the number of lines that contain the specified word. Test thoroughly

FOR ADVANCED STUDENTS (icount_words.py) Modify count_words.py to make the search case-insensitive.

Chapter 6: 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.

There is no one integrated tool for these libraries. You can either create scripts with your choice of IDE, or you can use iPython to manipulate data interactively and ad hoc.

NumPy overview

- Install numpy module from numpy.scipy.org
- 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.

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 a sequence of sequences.

The zeros() function expects a list 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 tuple of axis lenghts, and a value. It creates the array, putting the specified value in every element.

The empty() function creates an array initialized with random floats.

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

```
#!/usr/bin/env python
import numpy as np

a = np.array([[1, 2.1, 3], [4, 5, 6], [7, 8, 9], [20, 30, 40]])  ①
print(a)
print("# dims", a.ndim)  ②
print("shape", a.shape)  ③
print()

a_zeros = np.zeros((3, 5), dtype=np.uint32)  ④
print(a_zeros)
print()

a_ones = np.ones((2, 3, 4, 5))  ⑤
print(a_ones)
print()

# with uninitialized values
a_empty = np.empty((3, 8))  ⑥
print(a_empty)

print(a.dtype)  ⑦

nan_array = np.full((5, 10), np.NaN)  ⑧
print(nan_array)
```

① create array from nested sequences

② get number of dimensions

③ get shape

④ create array of specified shape and datatype, initialized to zeroes

⑤ create array of specified shape, initialized to ones

⑥ create uninitialized array of specified shape

⑦ defaults to float64

⑧ create array of NaN values

np_create_arrays.py

```
[[ 1.   2.1  3. ]
 [ 4.   5.   6. ]
 [ 7.   8.   9. ]
 [20.  30.  40. ]]
# dims 2
shape (4, 3)

[[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.]]]]

[[2.15433102e-314 2.15721303e-314 2.14462416e-314 2.14447563e-314
  2.68156159e+154 2.68156159e+154 2.14462878e-314 2.14505146e-314]
```

```
 [2.14447566e-314 2.14442468e-314 2.14454352e-314 2.14442055e-314
  2.14454355e-314 2.14442509e-314 2.14470833e-314 2.14472665e-314]
 [2.15721161e-314 2.15721164e-314 2.14618052e-314 2.14470924e-314
  2.14442187e-314 2.14444733e-314 2.15721171e-314 2.15721177e-314]]
float64
[[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().

The linspace() function creates a specified number of equally-spaced values.

The resulting arrays can be reshaped into multidimensional arrays.

Example

np_create_ranges.py

```
#!/usr/bin/env python
import numpy as np

r1 = np.arange(50)  ①
print(r1)
print("size is", r1.size)  ②
print()

r2 = np.arange(5, 101, 5)  ③
print(r2)
print("size is", r2.size)
print()

r3 = np.arange(1.0, 5.0, .3333333)  ④
print(r3)
print("size is", r3.size)
print()

r4 = np.linspace(1.0, 2.0, 10)  ⑤
print(r4)
print("size is", r4.size)
print()
```

① create range of ints from 0 to 49

② size is 50

③ create range of ints from 5 to 100 counting by 5

④ start, stop, and step may be floats

⑤ 10 equal steps between 1.0 and 2.0

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.

Some builtin array functions:

all, alltrue, any, apply along axis, argmax, argmin, argsort, average, bincount, ceil, clip, conj, conjugate, corrcoef, cov, cross, cumprod, cumsum, diff, dot, floor, inner, inv, lexsort, max, maximum, mean,median, min, minimum, nonzero, outer, prod, re, round, sometrue, sort, std, sum, trace, transpose, var, vdot, vectorize, where

Example

np_basic_array_ops.py

```
#!/usr/bin/env python
import numpy as np

a = np.array(
    [
        [5, 10, 15],
        [2, 4, 6],
        [3, 6, 9, ],
    ]
)  ①

b = np.array(
    [
        [10, 85, 92],
        [77, 16, 14],
```

```
        [19, 52, 23],
    ]
)  ②
print("a")
print(a)
print()
print("b")
print(b)
print()

print("a * 10")
print(a * 10)  ③
print()

print("a + b")
print(a + b)  ④
print()

print("b + 3")
print(b + 3)  ⑤
print()

s1 = a.sum()  ⑥
s2 = b.sum()  ⑥
print("sum of a is {0}; sum of b is {1}".format(s1, s2))
print()

a += 1000  ⑦
print(a)
```

① create 2D array

② create another 2D array

③ multiply every element by 10 (not in place)

④ add every element of a to the corresponding element of b

⑤ add 3 to every element of b

⑥ calculate sum of all elements

⑦ add 1000 to every element of a (in place)

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]]

sum of a is 60; sum of b is 388

[[1005 1010 1015]
 [1002 1004 1006]
 [1003 1006 1009]]
```

ufuncs and builtin operators

In normal Python, you are used to iterating over arrays, especially nested array, 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 on an array, it calls the underlying ufunc.

There are many other ufuncs built into numpy.

Shapes

- Number of elements on each axis
- array.shape has shape tuple
- Assign to array.shape to change
- array.ravel() to flatten to one dimension
- 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() method will flatten any array into a single dimension. The transpose() method will flip shape (x,y) to shape (y,x). It is equivalent to array.shape = reversed(array.shape).

Example

np_shapes.py

```
#!/usr/bin/env python
import numpy as np

a1 = np.arange(15)  ①
print("a1 shape", a1.shape)  ②
print()

print(a1)
print()

a1.shape = 3, 5  ③
print(a1)
print()

a1.shape = 5, 3  ④
print(a1)
print()

print(a1.flatten())  ⑤
print()

print(a1.transpose())  ⑥
print("-------------------")

a2 = np.arange(40)  ⑦
a2.shape = 2, 5, 4  ⑧

print(a2)
print()
```

① create 1D array

② get shape

③ reshape to 3x5

④ reshape to 5x3

⑤ print array as 1D

⑥ print transposed array

⑦ create 1D array

⑧ reshape go 2x5x4

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

```
#!/usr/bin/env 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]]
)  ①

print(a)
print()

print('a[0] =>', a[0])  ②
print('a[0][0] =>', a[0][0])  ③
print('a[0,0] =>', a[0, 0])  ④
print('a[0,:3] =>', a[0, :3])  ⑤
print('a[0,::2] =>', a[0, ::2])  ⑥
print()
print('a[::2] =>', a[::2])  ⑦
print()
print('a[:3, -2:] =>', a[:3, -2:])  ⑧
```

① sample data

② first row

③ first element of first row

④ same, but numpy style

⑤ first 3 elements of first row

⑥ every second element of first row

⑦ every second row

⑧ 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

```
#!/usr/bin/env 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]]
)  ①

print('a =>', a, '\n')

i = a > 50  ②
print('i (a > 50) =>', i, '\n')

print('a[i] =>', a[i], '\n')  ③

print('a[a > 50] =>', a[a > 50], '\n')  ④

print('a[i].min(), a[i].max() =>', a[i].min(), a[i].max(), '\n')  ⑤

a[i] = 0  ⑥
print('a =>', a, '\n')

print("a[a < 15] += 10")
a[a < 15] += 10  ⑦
print(a, '\n')
```

① sample data

② create Boolean mask

③ print elements of a that are > 50 using mask

④ same, but without creating a separate mask

⑤ min and max values of result set with values less than 50

⑥ set elements with value > 50 to 0

⑦ add 10 to elements < 15

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]]
```

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

```
#!/usr/bin/env python
import numpy as np

a = np.array(
    [[70, 31, 21, 76, 19, 5, 54, 66],
     [23, 29, 71, 12, 27, 74, 65, 73]]
)  ①

b = np.array(
    [[11, 84, 7, 10, 31, 50, 11, 98],
     [25, 13, 43, 1, 31, 52, 41, 90]]
)  ②

print('a =>\n', a)
print()
print('b =>\n', b)
print()
print('vstack((a,b)) =>\n', np.vstack((a, b)))  ③
print()

print('vstack((a,a[0] + a[1])) =>\n', np.vstack((a, a[0] + a[1])))  ④
print()

print('hstack((a,b)) =>\n', np.hstack((a, b)))  ⑤
```

① sample array a

② sample array b

③ stack arrays vertically (like pancakes)

④ add a row with sums of first two rows

⑤ 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]]
```

Iterating

- Similar to normal Python
- Iterates through first dimension
- Use array.flat to iterate through all elements

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 the flat property of the array.

NOTE Be aware that iterating over a NumPy array is generally much less efficient than using a *vectorized* approach: calling a ufunc or directly applying a math operator.

Example

np_iterating.py

```
#!/usr/bin/env python
import numpy as np

a = np.array(
    [[70, 31, 21, 76],
     [23, 29, 71, 12]]
)  ①

print('a =>\n', a)
print()

print("for row in a: =>")
for row in a:  ②
    print("row:", row)
print()

print("for column in a.T:")
for column in a.T:  ③
    print("column:", column)
print()

print("for elem in a.flat: =>")
for elem in a.flat:  ④
    print("element:", elem)
```

① sample array

② iterate over rows

③ iterate over columns by transposing the array

④ iterate over all elements (row-major)

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
```

Array creation shortcuts

- Use "magic" arrays r_, c_
- Either creates a new numpy array
- Index values determine resulting array
- List of arrays creates a "stacked" array
- List of values creates a 1D array
- Slice notation creates a range of values
- A complex step creates equally-spaced value

Numpy provides several shortcuts for working with arrays.

The r_ object can be used to magically build arrays via its index expression. It acts like a magic array, and "returns" (evaluates to) a normal numpy ndarray object.

There are two main ways to use r_():

If the index expression contains a list of arrays, then the arrays are "stacked" along the first axis.

If the index contains slice notation, then it creates a one-dimensional array, similar to numpy.arange(). It uses start, stop, and step values. However, if step is an imaginary number (a literal than ends with 'j'), then it specifies the number of points wanted, more like numpy.linspace().

There can be more than one slice, as well as individual values, and ranges. They will all be concatenated into one array.

The first element in the index can be a either a string that modifies how the array is created, or a string that makes the result a matrix instead of an ndarray.

If the first element is a string containing one, two, or three integers separated by commas, then the first integer is the axis to stack the arrays along; the second is the minimum number of dimensions to force each entry into; the third allows you to control the shape of the resulting array.

If the first element in the index is "r" or "c", then a numpy matrix object is returned. c_ works exactly like r_, but is column-oriented rather than row-oriented.

This is especially useful for making a new array from selected parts of an existing array.

NOTE Most of the time you will be creating arrays by reading data — these are mostly useful for edge cases when you're creating some smaller specialized array.

Example

np_tricks.py

```
#!/usr/bin/env python
import numpy as np

a1 = np.r_[np.array([1, 2, 3]), 0, 0, np.array([4, 5, 6])]  ①
print(a1)
print()

a2 = np.r_[-1:1:6j, [0] * 3, 5, 6]  ②
print(a2)
print()

a = np.array([[0, 1, 2], [3, 4, 5]])  ③
a3 = np.r_['-1', a, a]  ④
print(a3)
print()

a4 = np.r_['0,2', [1, 2, 3], [4, 5, 6]]  ④
print(a4)
print()

a5 = np.r_['0,2,0', [1, 2, 3], [4, 5, 6]]  ⑤
print(a5)
print()

a6 = np.r_['1,2,0', [1, 2, 3], [4, 5, 6]]  ⑥
print(a6)
print()

m = np.r_['r', [1, 2, 3], [4, 5, 6]]
print(m)
print(type(m))
```

① build array from a sequence of array-like things

② faux slice with complex step implements linspace <3>

np_tricks.py

```
[1 2 3 0 0 4 5 6]

[-1.  -0.6 -0.2  0.2  0.6  1.   0.   0.   0.   5.   6. ]

[[0 1 2 0 1 2]
 [3 4 5 3 4 5]]

[[1 2 3]
 [4 5 6]]

[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]

[[1 4]
 [2 5]
 [3 6]]

[[1 2 3 4 5 6]]
<class 'numpy.matrixlib.defmatrix.matrix'>
```

Matrices

- 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

```
#!/usr/bin/env python
import numpy as np

m1 = np.array(
    [[2, 4, 6],
     [10, 20, 30]]
)  ①

m2 = np.array([[1, 15],
               [3, 25],
               [5, 35]])  ②

print('m1 =>\n', m1)
print()

print('m2 =>\n', m2)
print()

print('m1 * 10 =>\n', m1 * 10)  ③
print()

print('m1 @ m2 =>\n', m1 @ m2)  ④
print()
```

① sample 2x3 array

② sample 3x2 array

③ multiply every element of m1 times 10

④ matrix multiply m1 times m2 — diagonal product

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

- Data type is inferred from initialization data
- Can be specified with arange(), ones(), zeros(), etc.

Numpy defines many different 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.

See https://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html for more details.

Example

np_data_types.py

```
#!/usr/bin/env python
import numpy as np

r1 = np.arange(45)  ①
r1.shape = (3, 3, 5)  ②
print('r1 datatype:', r1.dtype)
print('r1 =>\n', r1, '\n')

r2 = np.arange(45.)  ③
r2.shape = (3, 3, 5)
print('r2 datatype:', r2.dtype)
print('r2 =>\n', r2, '\n')

r3 = np.arange(45, dtype=np.int16)  ③
r3.shape = (3, 3, 5)
print('r3 datatype:', r3.dtype)
print('r3 =>\n', r3, '\n')
```

① create array — arange() defaults to int

② create array — passing float makes all elements float

③ create array — set datatype to short int

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]]]
```

List of NumPy routines by category

- Over 200 functions
- Doc strings and examples

Numpy has over 200 functions. To get more information, check out the NumPy Routines List at https://docs.scipy.org/doc/numpy/reference/routines.html.

This page has all of the Numpy functions and operators. Click on a routine to see the docstring and examples. It is a great resource for getting up to speed with Numpy.

Chapter 6 Exercises

Exercise 6-1 (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 6-2 (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 6-3 (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 7: 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 one, two, 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 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

Pandas architecture

- pandas.core.frame
- 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 indices (like the rows and columns of a spreadsheet, but more flexible).

You can specify the indices, or pandas will use successive integers. Each row or column of a DataFrame is a Series.

The third structure in pandas is a Panel, which is more more or less a collection of DataFrames, and describes three-dimensional (or higher) data.

Series

- Indexed list of values
- Similar to a dictionary, but ordered
- Can get sum(), mean(), etc.
- Use index to get individual values
- Indices 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 indices 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

```
#!/usr/bin/env python
import numpy as np
import pandas as pd

NUM_VALUES = 10
index = [chr(i) for i in range(97, 97 + NUM_VALUES)]  ①
print("index:", index, '\n')

s1 = pd.Series(np.linspace(1, 5, NUM_VALUES), index=index)  ②
s2 = pd.Series(np.linspace(1, 5, NUM_VALUES))  ③

print("s1:", s1, "\n")
print("s2:", s2, "\n")

print("selecting elements")
print(s1[['h', 'b']], "\n")  ④

print(s1[['a', 'b', 'c']], "\n")  ④

print("slice of elements")
print(s1['b':'d'], "\n")  ⑤

print("sum(), mean(), min(), max():")
print(s1.sum(), s1.mean(), s1.min(), s1.max(), "\n")  ⑥

print("cumsum(), cumprod():")
print(s1.cumsum(), s1.cumprod(), "\n")  ⑥

print('a' in s1)  ⑦
print('m' in s1)  ⑦
print()

s3 = s1 * 10  ⑧
print("s3 (which is s1 * 10)")
print(s3, "\n")

s1['e'] *= 5

print("boolean mask where s3 > 25:")
print(s3 > 25, "\n")  ⑨
```

```
print("assign -1 where mask is true")
s3[s3 < 25] = -1  ⑩
print(s3, "\n")

s4 = pd.Series([-0.204708, 0.478943, -0.519439])  ⑪
print("s4.max(), .min(), etc.")
print(s4.max(), s4.min(), s4.max() - s4.min(), '\n')  ⑫

s = pd.Series([5, 10, 15], ['a', 'b', 'c'])  ⑬
print("creating series with index")
print(s)
```

① make list of 'a', 'b', 'c', ...

② create series with specified index

③ create series with auto-generated index (0, 1, 2, 3, ...)

④ select items from series

⑤ select slice of elements

⑥ get stats on series

⑦ test for existence of label

⑧ create new series with every element of s1 multiplied by 10

⑨ create boolean mask from series

⑩ set element to -1 where mask is True

⑪ create new series

⑫ print stats

⑬ create new series with index

DataFrames

- Two-dimensional grid of values
- Row and column labels (indices)
- 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 6. 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 unioned 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 unioned 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

NOTE This utility method is used in some of the example scripts:

printheader.py

```
#!/usr/bin/env python
HEADER_CHAR = '='

def print_header(comment, header_width=50):
    ''' Print comment and separator '''
    header_line = HEADER_CHAR * header_width
    print(header_line)
    print(comment.center(header_width-2).center(header_width, HEADER_CHAR))
    print(header_line)

if __name__ == '__main__':
    print_header("this is a test")
```

Example

pandas_simple_dataframe.py

```
#!/usr/bin/env python
import pandas as pd
from printheader import print_header

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  ①
indices = ['a', 'b', 'c', 'd', 'e', 'f']  ②

values = [  ③
    [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)  ④
print_header('DataFrame df')
print(df, '\n')

print_header("df['gamma']")
print(df['gamma'])  ⑤
```

① column names

② row names

③ sample data

④ create dataframe with row and column names

⑤ 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
```

Data alignment

- pandas will auto-align data by rows and columns
- Non-overlapping data will be set as NaN

When two dataframes are combined, columns and indices are aligned.

The result is the union of matching rows and columns. Where data doesn't exist in one or the other dataframe, it is set to NaN.

A default value can be specified for the overlapping cells when combining dataframes with methods such as add() or mul().

Use the fill_value parameter to set a default for missing values.

Example

pandas_alignment.py

```
#!/usr/bin/env python
import numpy as np
import pandas as pd
from printheader import print_header  ①

dataset1 = np.arange(9.).reshape((3, 3))  ②

df1 = pd.DataFrame(  ③
    dataset1,
    columns=['apple', 'banana', 'mango'],
    index=['orange', 'purple', 'blue']
)

dataset2 = np.arange(12.).reshape((4, 3))  ②

df2 = pd.DataFrame(  ③
    dataset2,
    columns=['apple', 'banana', 'kiwi'],
    index=['orange', 'purple', 'blue', 'brown']
)

print_header('df1')
print(df1)  ④
print()

print_header('df2')
print(df2)  ④
print()

print_header('df1 + df2')
print(df1 + df2)  ⑤

print_header('df1.add(df2, fill_value=0)')
print(df1.add(df2, fill_value=0))  ⑥
```

① provided with lab files to make output easier to read

② create a numpy array

③ create second Pandas dataframe from dataset, adding row and column numbers; note rows and columns do not quite match

④ output dataframe

⑤ when dataframes are combined, if no match for row + column label, set value to NaN

⑥ same as #5, but where one dataframe has a value, set it to 0

pandas_alignment.py

```
================================================
=                     df1                      =
================================================
        apple  banana  mango
orange    0.0     1.0    2.0
purple    3.0     4.0    5.0
blue      6.0     7.0    8.0

================================================
=                     df2                      =
================================================
        apple  banana  kiwi
orange    0.0     1.0   2.0
purple    3.0     4.0   5.0
blue      6.0     7.0   8.0
brown     9.0    10.0  11.0

================================================
=                   df1 + df2                  =
================================================
        apple  banana  kiwi  mango
blue     12.0    14.0   NaN    NaN
brown     NaN     NaN   NaN    NaN
orange    0.0     2.0   NaN    NaN
purple    6.0     8.0   NaN    NaN
================================================
=           df1.add(df2, fill_value=0)         =
================================================
        apple  banana  kiwi  mango
blue     12.0    14.0   8.0    8.0
brown     9.0    10.0  11.0    NaN
orange    0.0     2.0   2.0    2.0
purple    6.0     8.0   5.0    5.0
```

Index objects

- Used to index Series or DataFrames
- index = pandas.core.frame.Index(sequence)
- Can be named

An index object is a kind of ordered set that is used to access rows or columns in a dataset. As shown earlier, indexes can be specified as lists or other sequences when creating a Series or DataFrame.

You can create an index object and then create a Series or a DataFrame using the index object. Index objects can be named, either something obvious like 'rows' or columns', or more appropriate to the specific type of of data being indexed.

Remember that index objects act like sets, so the main operations on them are unions, intersections, or differences.

TIP You can replace an existing index on a DataFrame with the set_index() method.

Example

pandas_index_objects.py

```
#!/usr/bin/env python
import pandas as pd
from printheader import print_header

index1 = pd.Index(['a', 'b', 'c'], name='letters')  ①
index2 = pd.Index(['b', 'a', 'c'])
index3 = pd.Index(['b', 'c', 'd'])
index4 = pd.Index(['red', 'blue', 'green'], name='colors')

print_header("index1, index2, index3", 70)  ②
print(index1)
print(index2)
print(index3)
print()

print_header("index2 & index3", 70)
# these are the same
```

```
print(index2 & index3)  ③
print(index2.intersection(index3))  ③
print()

print_header("index2 | index3", 70)
# these are the same
print(index2 | index3)  ④
print(index2.union(index3))
print()

print_header("index1.difference(index3)", 70)
print(index1.difference(index3))  ⑤
print()

print_header("Series([10,20,30], index=index1)", 70)
series1 = pd.Series([10, 20, 30], index=index1)  ⑥
print(series1)
print()

print_header("DataFrame([(1,2,3),(4,5,6),(7,8,9)], index=index1, columns=index4)",
70)
dataframe1 = pd.DataFrame([(1, 2, 3), (4, 5, 6), (7, 8, 9)], index=index1,
columns=index4)
print(dataframe1)
print()

print_header("DataFrame([(1,2,3),(4,5,6),(7,8,9)], index=index4, columns=index1)",
70)
dataframe2 = pd.DataFrame([(1, 2, 3), (4, 5, 6), (7, 8, 9)], index=index4,
columns=index1)
print(dataframe2)
print()
```

① create some indexes

② display indexes

③ get intersection of indexes

④ get union of indexes

⑤ get difference of indexes

⑥ use index with series (can also be used with dataframe)

pandas_index_objects.py

```
========================================================================
=                       index1, index2, index3                         =
========================================================================
Index(['a', 'b', 'c'], dtype='object', name='letters')
Index(['b', 'a', 'c'], dtype='object')
Index(['b', 'c', 'd'], dtype='object')

========================================================================
=                           index2 & index3                            =
========================================================================
Index(['b', 'c'], dtype='object')
Index(['b', 'c'], dtype='object')

========================================================================
=                           index2 | index3                            =
========================================================================
Index(['a', 'b', 'c', 'd'], dtype='object')
Index(['a', 'b', 'c', 'd'], dtype='object')

========================================================================
=                      index1.difference(index3)                       =
========================================================================
Index(['a'], dtype='object')

========================================================================
=                   Series([10,20,30], index=index1)                   =
========================================================================
letters
a    10
b    20
c    30
dtype: int64

========================================================================
= DataFrame([(1,2,3),(4,5,6),(7,8,9)], index=index1, columns=index4) =
========================================================================
colors   red  blue  green
letters
a          1     2      3
b          4     5      6
c          7     8      9
```

```
=========================================================================
= DataFrame([(1,2,3),(4,5,6),(7,8,9)], index=index4, columns=index1) =
=========================================================================
letters  a  b  c
colors
red      1  2  3
blue     4  5  6
green    7  8  9
```

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

```
#!/usr/bin/env python
import pandas as pd
from printheader import print_header

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  ①
index = ['a', 'b', 'c', 'd', 'e', 'f']  ②

values = [  ③
    [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)  ④
print_header('DataFrame df')
print(df, '\n')

print_header("df['alpha']")
print(df['alpha'], '\n')  ⑤

print_header("df.beta")
print(df.beta, '\n')  ⑥

print_header("df['b':'e']")
print(df['b':'e'], '\n')  ⑦

print_header("df[['alpha','epsilon','beta']]")
print(df[['alpha', 'epsilon', 'beta']])  ⑧
print()

print_header("df[['alpha','epsilon','beta']]['b':'e']")
print(df[['alpha', 'epsilon', 'beta']]['b':'e'])  ⑨
print()
```

① column labels

② row labels

③ sample data

④ create dataframe with data, row labels, and column labels

⑤ select column 'alpha'

⑥ same, but alternate syntax (only works if column name is letters, digits, and underscores)

⑦ select rows 'b' through 'e' using slice of row labels

⑧ select columns — note index is an iterable

⑨ select columns AND slice rows

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['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[['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[['alpha','epsilon','beta']]['b':'e']     =
==================================================
   alpha  epsilon  beta
b    200      240   210
c    300      340   310
d    400      440   410
e    500      540   510
```

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

```
#!/usr/bin/env python
import pandas as pd
from printheader import print_header

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  ①
index = pd.date_range('2013-01-01 00:00:00', periods=6, freq='D')  ②

print(index, "\n")

values = [  ③
    [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, index, cols)  ④
print_header("Basic DataFrame:")
print(df)
print()

print_header("Triple each value")
print(df * 3)
print()  ⑤

print_header("Multiply column gamma by 1.5")
df['gamma'] *= 1.5  ⑥
print(df)
print()
```

① column labels

② date range to be used as row indexes

③ sample data

④ create dataframe from data

⑤ multiply every value by 3

⑥ multiply values in column 'gamma' by 1.

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
```

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

```
#!/usr/bin/env python
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)  ①
print_header('DataFrame df')
print(df, '\n')

df2 = df.drop(['beta', 'delta'], axis=1)  ②
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'])  ③
print(df3)
```

① create dataframe

② drop columns beta and delta (axes: 0=rows, 1=columns)

③ drop rows b, c, and e

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
```

Time Series

- Use time_series()
- Specify start/end time/date, number of periods, time units
- Useful as index to other data
- freq=time_unit
- periods=number_of_periods

pandas provides a function time_series() to generate a list of timestamps. You can specify the start/end times as dates or dates/times, and the type of time units. Alternatively, you can specify a start date/time and the number of periods to create.

The frequency strings can have multiples – 5H means every 5 hours, 3S means every 3 seconds, etc.

Table 7. Units for time_series() freq flag

Unit	**Represents**
M	Month
D	Day
H	Hour
T	Minute
S	Second

Example

pandas_time_slices.py

```
#!/usr/bin/env python
import pandas as pd
import numpy as np

hourly = pd.date_range('1/1/2013 00:00:00', '1/3/2013 23:59:59', freq='H')  ①
print("Number of periods: ", len(hourly))

seconds = pd.date_range('1/1/2013 12:00:00', freq='S', periods=(60 * 60 * 18))  ②
print("Number of periods: ", len(seconds))
print("Last second: ", seconds[-1])

monthly = pd.date_range('1/1/2013', '12/31/2013', freq='M')  ③
print("Number of periods: {0} Seventh element: {1}".format(
    len(monthly),
    monthly[6]
))

NUM_DATA_POINTS = 1441  ④

dates = pd.date_range('4/1/2013 00:00:00', periods=NUM_DATA_POINTS, freq='T')  ⑤

data = np.random.random(NUM_DATA_POINTS)  ⑥

series = pd.Series(data, index=dates)  ⑦

time_slice = series['4/1/2013 10:00:00':'4/1/2013 10:30:00']  ⑧
print(time_slice)  # 31 values
```

① make time series — every hour for 3 days

② make time series — every second for 18 hours

③ every month for 1 year

④ number of minutes in a day

⑤ create range from starting point with specified number of points — one day's worth of minutes

⑥ a day's worth of data

⑦ series indexed by minutes

⑧ select the half hour of data from 10:00 to 10:30

pandas_time_slices.py

```
Number of periods:  72
Number of periods:  64800
Last second:  2013-01-02 05:59:59
Number of periods: 12 Seventh element: 2013-07-31 00:00:00
2013-04-01 10:00:00    0.940646
2013-04-01 10:01:00    0.096089
2013-04-01 10:02:00    0.248606
2013-04-01 10:03:00    0.789897
2013-04-01 10:04:00    0.114204
2013-04-01 10:05:00    0.965126
2013-04-01 10:06:00    0.185748
2013-04-01 10:07:00    0.982763
2013-04-01 10:08:00    0.740299
2013-04-01 10:09:00    0.932912
2013-04-01 10:10:00    0.117964
2013-04-01 10:11:00    0.075580
2013-04-01 10:12:00    0.597483
2013-04-01 10:13:00    0.070609
2013-04-01 10:14:00    0.977615
2013-04-01 10:15:00    0.107569
2013-04-01 10:16:00    0.426324
2013-04-01 10:17:00    0.455133
2013-04-01 10:18:00    0.827247
2013-04-01 10:19:00    0.061402
2013-04-01 10:20:00    0.363280
2013-04-01 10:21:00    0.692623
2013-04-01 10:22:00    0.687107
2013-04-01 10:23:00    0.232886
2013-04-01 10:24:00    0.308176
2013-04-01 10:25:00    0.383642
2013-04-01 10:26:00    0.146308
2013-04-01 10:27:00    0.707394
2013-04-01 10:28:00    0.092515
2013-04-01 10:29:00    0.765656
2013-04-01 10:30:00    0.033086
Freq: T, dtype: float64
```

Table 8. Methods and attributes for fetching 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.values	S.values
Get the actual values from a data structure	DF.loc[row_indexer, col_indexer]
Multi-axis indexing by label (not by position)	DF.iloc[row_indexer, col_indexer]
Multi-axis indexing by integer position (not by labels)	DF.ix[row_indexer, col_indexer]

Table 9. Methods for Computations/Descriptive Stats

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.

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 support many different input formats. It will reading file headings and use them to create column indexes. By default, it will use integers for row indices, but you can specify a column to use as the index.

The read methods 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.

NOTE See Jupyter notebook **Pandas_Input_Demo** for examples of reading most types of input.

Table 10. Pandas I/O methods

Methods	Reads the following into a DataFrame
read_table()	Generic delimited file
read_csv()	CSV file
read_fwf()	File with fixed-width fields
read_clipboard()	Data from OS clipboard (passed to read_table())
read_excel()	Excel table
read_html()	HTML table
read_hdf()	HDF5 Store (using PyTables)
read_sql_query()	Result of SQL query

Example

pandas_read_csv.py

```
#!/usr/bin/env python
import pandas as pd
from printheader import print_header

# data from
# http://www.rita.dot.gov/bts/sites/rita.dot.gov.bts/files/publications/
# national_transportation_statistics/html/table_01_44.html

airports_df = pd.read_csv('../DATA/airport_boardings.csv', thousands=',',
index_col=1)  ①

print_header("HEAD OF DATAFRAME")

print(airports_df.head(), "\n")

print_header("SELECTED COLUMNS WITH FILTERED ROWS")
columns_wanted = ['2001 Total', 'Airport']
sort_col = '2001 Total'
max_val = 20000000
selector = airports_df['2001 Total'] > max_val
selected = airports_df[selector][columns_wanted]
print(selected)

print_header("COLUMN TOTALS")
print(airports_df[['2001 Total', '2010 Total']].sum(), "\n")

# print_header("'CODE' COLUMN SET AS INDEX")
# airports_df.set_index('Code')
# print(airports_df)

print_header("FIRST FIVE ROWS")
print(airports_df.iloc[:5])
```

① Read CSV file into dataframe; parse numbers containing commas,use first column as row index.

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 7 Exercises

Exercise 7-1 (simple_dataframe.py)

Create a DataFrame with columns named 'Test1', 'Test2', up to 'Test6'. Use default row indexes. Fill the DataFrame with random values.

- Print only columns 'Test3' and 'Test5'.
- Print the dataframe with every value multiplied by 3.6

Exercise 7-2 (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 8: 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 8 Exercises

Exercise 8-1 (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"
,"1995","1996","1997","1998","1999","2000","2001","2002","2003","2004","2005","2006"
,"2007","2008","2009","2010","2011"]
```

TIP See the script pandas_energy.py in the EXAMPLES folder to see how to load the data.

Chapter 9: Using Flask-SQLAlchemy

Objectives

- Integrate SQLAlchemy with Flask
- Define models
- Create database
- Access data from views

Using Flask-SQLAlchemy

- Create models (classes that map to DB tables)
- Generates SQL as needed
- Can create DB from scratch
- Interact with DB via objects, not SQL

SQL-Alchemy is a popular choice for supporting Flask apps. It is relatively easy to use, and works with SQLite3, MySQL, PostgreSQL, Oracle, Microsoft SQL Server, and others.

It works by using class-based *models*, which map to the database tables.

Creating models

- Create class that inherits from db.Model
- Add columns as class variables
- Add str() or repr() methods as desired
- Add constructor (*init*) to populate the model

To create a model in SQLAlchemy, define a class that inherits from the Model base class. To add columns, add class variables of type Column. The arguments to Column define the data type and other properties of the column

Add a *repr* method for displaying the raw model.

Creating the database

- Call create_all from the DB object

Once the models are defined, call create_all() to generate and execute the SQL to build the initial database.

Example

flask_sa_models.py

```
#!/usr/bin/env python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///../DATA/presidents.db'
db = SQLAlchemy(app)


class President(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    firstname = db.Column(db.String(80))
    lastname = db.Column(db.String(80))
    birthstate = db.Column(db.String(2))

    def __init__(self, firstname, lastname, state):
        self.firstname = firstname
        self.lastname = lastname
        self.birthstate = state

    def __repr__(self):
        return '<President {} {}>'.format(self.firstname, self.lastname)


def create_db():
    '''Warning! Only run this once for production!!!!'''
    db.create_all()


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

Adding and accessing data

- To use the database, create instances of models
- Add model instances to db session with db.session.add()
- Call session.commit() to save all added instances

To add data to the database, create an instance of a model, and populate it with appropriate data. Add the instance to the database session. When one or more instances have been added, use db.session.commit() to save all instances added since the last commit.

Example

flask_sa_main.py

```
#!/usr/bin/env python
from flask import Flask
# from flask.ext.sqlalchemy import SQLAlchemy
from flask_sa_models import db, President

app = Flask(__name__)

@app.route('/president/<int:term>')
def add_pres(term):
    # get user input here from form or API or wherever....
    president = President('Abraham', 'Lincoln', 'IL')
    db.session.add(president)
    db.session.commit()

    return '''<h1>Added.</h1>''', 201


if __name__ == '__main__':
    app.run(debug=True)
```

Building and using queries

- Create select objects
- Equivalent to SQL 'select' statements
- Can select from more than one table
- Can add filters, ordering, etc.
- Call select() from table
- Call execute() from query
- Execute returns an iterable result

From a Table object, you can call the select() method to create a query object. This query can then be executed, via the aptly-named execute() method.

Fetching results

- All queries start with session
- Use query() to get a query object
- Call methods on query object

To get data from the database, use the session object call the **query()** method, passing in the name of the object.

From that object, you can call **all()**, **count()**, or other query methods.

Example

flask_sa_movie_search_db.py

```
#!/usr/bin/env python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

from sa_movie_models import Movie, Director


def main():
    session = setup()
    do_query_1(session)
    print()
    do_query_2(session)
    session.close()


def setup():
    engine = create_engine(
        'sqlite:///../DATA/movies.db',
        echo=False
    )
    SESSION = sessionmaker(bind=engine)
    return SESSION()


def do_query_1(session):
```

```
    for director in session.query(Director):
        if director.first_name == 'Steven':
            director.first_name = 'Bob'
        print(director.first_name, director.last_name)
        for movie in director.movies:
            print("\t{0} ({1})".format(movie.movie_name, movie.year))


def do_query_2(session):
    movie_name_query = 'M*A*S*H'

    # select * from Movie where movie_name  = 'M*A*S*H'
    for result in session.query(Movie).filter(Movie.movie_name == movie_name_query):
        print("{0} was directed by {1} {2}".format(
            movie_name_query,
            result.person.first_name,
            result.person.last_name,
        ))


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

Table 11. SQLAlchemy Query Functions

all()	Returns all the objects in the query
count()	Returns the total number of rows of a query
filter()	Filters the query by applying a criteria
delete()	Removes from the database the rows matched by a query
distinct()	Applies a distinct statement to a query
exists()	Adds an exists operator to a sub-query
first()	Returns the first row in a query
get()	Returns the row referenced by the primary key parameter passed as argument
join()	Creates a SQL join in a query
limit()	Limits the number of rows returned by a query
order_by()	Sets an order in the rows returned by a query

Chapter 9 Exercises

Exercise 9-1 (knight_app.py)

Using the knight table from the previous chapter, create a Flask app that searches the knight database by name.

Chapter 10: An Overview of Flask

Objectives

- Understand what Flask is
- Understand what Flask isn't
- Review Flask's basic architecture
- Learn some of Flask's features

What is Flask?

- Microframework for web development
- Python package
- Provides
  - URL mapping
  - Templating (Jinja2)
  - RESTful request dispatching

Flask is a Python microframework for web development.

With flask you can develop large or small web applications, as well as provide web services via REST.

Flask provides basic functionality for web development, but does not provide such features as **DBMS** Authentication ** Form validation

The developer can use any Python packages for these features.

Origins and purpose

- Written by Armin Ronacher
- Pocoo project
- Based on Werkzeug and Jinja2

Flask was created by Armin Ronacher as a Pocoo project. It is based on Werkzeug, a WSGI package, and Jinja2, a template package.

It was originally developed as a tool for creating a bulletin board system.

The creators of Flask also created Werkzeug, Jinja2, Pygments, Sphinx, and other web tools.

NOTE "Pocoo" is not a real word – Ronacher and Georg Brandl made it up

Microframework vs Framework

- Framework provides complete structure for app
- Framework makes most decisions
- Microframework provides core facilities
- Microframework does not make most decisions

Flask is a microframework. It provides core facilities for developing web applications.

The developers made a conscious decision to keep Flask small. The "micro" part means that Flask itself is small, compared to frameworks such as Django or Pyramid, not that you can’t create large applications.

Django and Pyramid generate the infrastructure for web applications, and the developer fills in details in predefined scripts. Both frameworks provide ORM tools, templates, authentication, session, and other components. Although other tools can be used, it is more work to stray beyond the predefined components.

Flask does not generate any infrastructure. The developer creates scripts that use Flask as needed. Any appropriate components may be used for databases, forms, security, sessions, etc.

Views, controllers, but no models

- Flask provides views via Jinja2 templates
- Flask provides controllers via decorated functions
- Flask does not provide models

The term Model-View-Controller (MVC) is often used to describe the architecture of web applications.

The Model is the raw data that will be used to provide a response to the user. Most web apps that need models implement them as SQL (or possibly no-SQL) databases. Flask does not provide an integrated database package.

The View is the finished resource that is returned to the client (browser or REST client). Views are implemented via Jinja2 templates by default, but any template package can be used.

The Controller is the business logic that turns the Model into the View. In Flask, controllers are implemented as functions, These functions use decorates to map to routes, and return Views (content).

Extensions

- Packages designed to integrate with Flask
- Extend Flask without bloating the core

For convenience, there are many extensions to Flask that integrate useful components into flask. These extensions act as front-ends to many existing packages.

They allow Flask to remain small and flexible, while providing extensive functionality.

Some of the most popular extensions are:

- Flask-Admin
- Flask-Cache
- Flask-Login
- Flask-OAuth
- Flask-SQLAlchemy
- Flask-WTF

To see a complete list, go to http://flask.pocoo.org/extensions/.

Licensing

- BSD

Flask is licensed under the BSD License.

Who's using Flask?

- Pinterest https://www.pinterest.com/
- Twilio https://www.twilio.com/
- LinkedIn http://www.linkedin.com

Some large web sites are implemented with Flask.

Steve Cohen, the API guy at Pinterest, says:

Since early 2012, we've reinvested in the API, putting it on equal footing with the web and since late 2012 every Pinterest client (web, iOS or Android) hits it at some point. Presently, the API does over 12 billion requests per day, all in Flask.

Flask is server-side software

- Provides
  - Routing
  - Business Logic
  - HTTP Responses
- Part of a stack with many layers

It's import to remember that Flask is a framework that provides specific tasks, including routing and HTTP responses (which can be page views or RESTful data). IT is not a complete Web solution.

In addition to Flask, you will need expertise in HTML, CSS, JavaScript, as well as Python.

You will also need to be at least somewhat familiar with a WSGI-compatible web server for deployment.

If your application requires permanent data storage, you will need to understand how to use SQL with a relational database, or one of the alternatives, such as NoSQL, or some other data store.

Chapter 11: Implementing REST with Flask

Objectives

- Learn what REST means
- Implement a REST API
- Write view functions to respond to REST requests

What is REST?

- **REpresentational State Transfer**
- Architectural style, not protocol
- Programmatic access to data via web server
- Replaces older protocols (e.g. SOAP)

REST is an architectural style for providing web-based services. It is not a protocol. It replaces older styles, such as RPC and SOAP.

REST specifies six constraints which must be met in order to be called RESTful. Since it is a guideline and not a protocol, all requirements do not necessarily have to be met.

The commonly recognizable part of REST is that typical REST servers are accessed by sending URL path information in the request, rather than URL queries.

In other words, a traditional request might look like http://yourserver.com/presidents?term=26 The REST equivalent might look like http://yourserver.com/presidents/term/26 or just http://yourserver.com/presidents/26

Your application's RESTful API is defined in terms of such URLs.

Base REST constraints

The base constraints of a web-based REST system are:

- Client-server architecture via HTTP
- Stateless communication (each request is independent of the last)
- Cacheable
- Layered system
- Code on demand (optional)
- Uniform interface
    - Hypertext (HTML) links to reference state (i.e., retrieve data)
    - Hypertext links to get related resource
    - Standard media types (typically JSON, but sometimes XML or others)

Designing a REST API

- It's all about the URL
- Include version in paths
- Human-readable

When designing a RESTful API, it's all about the URL. All communication from the client to the server is done in terms of URL requests. This includes queries (GET) as well as updates (POST, PUT) and deletes (DELETE).

Because a service has no control over the clients that are using it, a service cannot abruptly change the API. For this reason, you should include the API version in URLs. Once you have announced an API change, the client can update its code.

For instance, the first version of a site might use the URL

```
http://myserver.com/api/v1.0/president/term/26
```

But the next version might change it to just

```
http://myserver.com/api/v2.0/president/26
```

This allows the user to present the new version while keeping the old version available.

URLs in your API should be as short as possible while remaining descriptive and human-readable.

Example

```
http://myserver.com/api/v2.0/presidents
http://myserver.com/api/v2.0/president/26
http://myserver.com/api/v2.0/presidents/roosevelt
```

If the service allows data to be modified, rather than reflect this in the URL, it is controlled by HTTP verbs. Instead of providing deletion with

```
http://myserver.com/api/v2.0/president/delete/26
```

provide it with the same URL as the query, but with the client providing a DELETE request, rather than GET:

```
http://myserver.com/api/v2.0/president/26
```

Here is a good presentation on the design of APIs:

```
https://speakerdeck.com/dzuelke/designing-http-interfaces-and-restful-web-services-
sfliveparis2012-2012-06-08
```

Setting up RESTful routes

*Use view functions *Parameter types specified with converters *Path information passed into view functions

To set up a RESTful URL, create a view function as usual. In the route string, enclose one or more parameters in angle brackets, like

```
/president/<int:term>
```

The word before the colon is a converter, and is used to parse a data type from the path information.

The view function's parameter names must match the names in the route:

```
def president_by_term(term):
    pass
```

There can be any number of path parameters in a URL.

Choosing the return type

- Return type is typically JSON
- Other types requested via HTTP header
- Put logic in view function, or use decorator

RESTful services typically return data as JSON (JavaScript Object Notation). However, the client may request another data type via the HTTP ACCEPT header. You can decide to provide alternate data types, or just return an error status if that data type is not available.

Example

flask_rest_response.py

```
#!/usr/bin/env python
#
from datetime import date
import xml.etree.ElementTree as ET
from flask import Flask, request, render_template, jsonify
from flask_bootstrap import Bootstrap

from president import President

app = Flask(__name__)
Bootstrap(app)


def pres_to_dict(pres):
    """Convert one president object to a dictionary"""
    pres_dict = {}
    for prop_name in dir(pres):
        if not prop_name.startswith('_'):
            prop_value = getattr(pres, prop_name)
            if isinstance(prop_value, date):
                prop_value = '{0.year:4d}-{0.month:02d}-
{0.day:02d}'.format(prop_value)
            pres_dict[prop_name] = prop_value
    return pres_dict
```

```
def pres_list_to_xml(pres_list):
    """Convert list of presidents to XML"""
    root_element = ET.Element('presidents')
    for pres in pres_list:
        pres_element = ET.Element('president')
        fname = ET.Element('firstname')
        fname.text = pres.first_name
        pres_element.append(fname)
        lname = ET.Element('lastname')
        lname.text = pres.last_name
        pres_element.append(lname)
        root_element.append(pres_element)
    return ET.tostring(root_element)

@app.route('/api/1.0/potus')
def index():
    """Main (and only) page; returns list of all presidents"""
    presidents = []
    for i in range(1, 46):
        presidents.append(President(i))
    accept_type = request.headers.get('ACCEPT')
    response = ''
    print("Accept type:", accept_type)
    if accept_type.startswith('text/html'):
        response = render_template('president_list_bs.html', presidents=presidents)
    elif accept_type.startswith('application/xml'):
        response = pres_list_to_xml(presidents)
    elif accept_type == 'application/json':
        presidents_as_dicts = [pres_to_dict(p) for p in presidents]
        response = jsonify(presidents=presidents_as_dicts)
    # handle error here if non-expected type

    return response



if __name__ == '__main__':
    app.run(debug=True)
```

Using jsonify

- jsonify() creates JSON suitable for the response
- Pass keyword arguments which becomes keys in JSON dictionary flask.jsonify() returns a JSON response created from named parameters.

Example

```
from flask import jsonify

...

    return jsonify(movie='Jaws', directory='Spielberg')
```

Handling invalid requests

- Use app.errorhandler()

Flask's builtin exceptions typically return human-friendly HTML. This is great for humans, but not so great for API clients.

The solution is to define a custom exception type and install an error handler for it.

Example

```
class InvalidAPIRequest(Exception):
    status_code = 400

    def __init__(self, message, status_code=None, data=None):
        Exception.__init__(self)
        self.message = message
        if status_code is not None:
            self.status_code = status_code
        self._data = data

    def to_dict(self):
        return_value = dict(self.data or ())
        return_value['message'] = self.message
        return  return_value

@app.errorhandler(InvalidAPIRequest)
def handle_invalid_usage(error):
    response = jsonify(error.to_dict())
    response.status_code = error.status_code
    return response

@app.route('/president/<int:term>')
def get_president(term):
    if term < 1 or term > 44:
        raise InvalidUsage('Term out of range', status_code=410)
```

Receiving data from POST

- Use the request.form object
- form is a dictionary of fields

To receive data from a form, use a route that specifies a POST request, and use the request.form object to retrieve specific fields.

Example

```
@app.route('/president', methods=['POST'])
def add_president():
    first_name = request.form['firstname']
    last_name = request.form['lastname']
```

Handling a DELETE request

- No special URL
- Use route with DELETE method

To handle a delete request, use a route that specifies the DELETE method.

```
@app.route('/president/<int:term>', methods=['DELETE'])
def delete_president(term):
    pass   # delete president from database here...
```

Chapter 11 Exercises

Exercise 11-1 (restful_knights.py)

Write a script to provide access to knight data via a RESTful interface. Design a simple API.

You can use the Knight class from knight.py for data.

```
/knight/name  (to get one knight)
```

or

```
/knights  (to get a list of all knights)
```

Return the data as JSON. Use postman or a similar tool to test (or just use a browser).

Chapter 12: 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

Grabbing a web page

- import urlopen from urllib.request
- urlopen() similar to open()
- Iterate through (or read from) response
- Use info() method for metadata

The standard library module **urllib.request** includes **urlopen()** for reading data from web pages. urlopen() 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.

Example

read_html_urllib.py

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

import urllib.request

u = urllib.request.urlopen("https://www.python.org")

print(u.info())  ①
print()

print(u.read(500).decode())   ②
```

① .info() returns a dictionary of HTTP headers

② 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: 49044
Server: nginx
Content-Type: text/html; charset=utf-8
X-Frame-Options: DENY
Via: 1.1 vegur
Via: 1.1 varnish
Accept-Ranges: bytes
Date: Wed, 19 Feb 2020 17:04:53 GMT
Via: 1.1 varnish
Age: 115
X-Served-By: cache-bwi5123-BWI, cache-lga21942-LGA
X-Cache: HIT, HIT
X-Cache-Hits: 1, 1
X-Timer: S1582131893.403758,VS0,VE1
Vary: Cookie
Strict-Transport-Security: max-age=63072000; includeSubDomains



<!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>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <link rel="prefetch" href="//ajax.googleapis.com/ajax/libs/jqu
```

Example

read_pdf_urllib.py

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

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'  ①

saved_pdf_file = 'nasa_iss.pdf'  ②

try:
    URL = urlopen(url)  ③
except HTTPError as e:  ④
    print("Unable to open URL:", e)
    sys.exit(1)

pdf_contents = URL.read()  ⑤
URL.close()

with open(saved_pdf_file, 'wb') as pdf_in:
    pdf_in.write(pdf_contents)  ⑥

if sys.platform == 'win32':  ⑦
    cmd = saved_pdf_file
elif sys.platform == 'darwin':
    cmd = 'open ' + saved_pdf_file
else:
    cmd = 'acroread ' + saved_pdf_file

os.system(cmd)  ⑧
```

① target URL

② name of PDF file for saving

③ open the URL

④ catch any HTTP errors

⑤ read all data from URL in binary mode

⑥ write data to a local file in binary mode

⑦ select platform and choose the app to open the PDF file

⑧ launch the app

Consuming Web services

- 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

```
#!/usr/bin/env python
"""
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'

API_KEY = 'b619b55d-faa3-442b-a119-dd906adc79c8'

URL_TEMPLATE =
'https://www.dictionaryapi.com/api/v3/references/collegiate/json/{}?key={}'  ①

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)  ②

    do_query(url)

def do_query(url):
    print("URL:", url)
    request = Request(url)
    response = urlopen(request)  ③
    raw_json_string = response.read().decode()  ④
    data = json.loads(raw_json_string)  ⑤
    # print("RAW DATA:")
    # pprint(data)
    for entry in data:  ⑥
        if isinstance(entry, dict):
            meta = entry.get("meta") ⑦
            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:])
```

① base URL of resource site

② build search URL

③ send HTTP request and get HTTP response

④ read content from web site and decode() from bytes to str

⑤ convert JSON string to Python data structure

⑥ iterate over each entry in results

⑦ retrieve items from results (JSON convert to lists and dicts)

web_content_consumer_urllib.py dewars

```
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
```

HTTP the easy way

- Use the **requests** module
- Pythonic front end to urllib, urllib2, httplib, etc
- Makes HTTP transactions simple

While **urllib** and friends are powerful libraries, their interfaces are complex for non-trivial tasks. You have to do a lot of work 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 Anaconda, or is readily available from PyPI via **pip**.

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.

Example

read_html_requests.py

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

import requests

response = requests.get("https://www.python.org")  ①

for header, value in sorted(response.headers.items()): ②
    print(header, value)
print()

print(response.content[:200].decode())   ③
print('...')
print(response.content[-200:].decode())   ④
```

① requests.get() returns HTTP response object

② response.headers is a dictionary of the headers

③ The text is returned as a bytes object, so it needs to be decoded to a string; print the first 200 bytes

④ print the last 200 bytes

Example

read_pdf_requests.py

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

import sys
import os

import requests

url = 'https://www.nasa.gov/pdf/739318main_ISS%20Utilization%20Brochure
%202012%20Screenres%203-8-13.pdf'  ①
saved_pdf_file = 'nasa_iss.pdf'  ②

response = requests.get(url)  ③
if response.status_code == requests.codes.OK:  ④

    with open(saved_pdf_file, 'wb') as pdf_in:  ⑤
        pdf_in.write(response.content)  ⑥

    if sys.platform == 'win32':  ⑦
        cmd = saved_pdf_file
    elif sys.platform == 'darwin':
        cmd = 'open ' + saved_pdf_file
    else:
        cmd = 'acroread ' + saved_pdf_file

    os.system(cmd)  ⑧
```

① target URL

② name of PDF file for saving

③ open the URL

④ check status code

⑤ open local file

⑥ write data to a local file in binary mode; response.content is data from URL

⑦ select platform and choose the app to open the PDF file

⑧ launch the app

Example

web_content_consumer_requests.py

```
#!/usr/bin/env python
from pprint import pprint
import sys
import requests


BASE_URL = 'https://www.dictionaryapi.com/api/v3/references/collegiate/json/'  ①

API_KEY = 'b619b55d-faa3-442b-a119-dd906adc79c8' ②


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},
    )  ③

    if response.status_code == requests.codes.OK:
        # pprint(response.content.decode())
        # print('-' * 60)
        data = response.json()  ④
        for entry in data: ⑤
            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:])
```

① base URL of resource site

② credentials

③ send HTTP request and get HTTP response

④ convert JSON content to Python data structure

⑤ check for results

web_content_consumer_requests.py "maker's mark"

```
WOMBAT (noun)
any of several stocky burrowing Australian marsupials (genera Vombatus and
Lasiorhinus of the family Vombatidae) resembling small bears
```

TIP See details of requests API at http://docs.python-requests.org/en/v3.0.0/api/#main-interface

Table 12. 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

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

```
#!/usr/bin/env python
from getpass import getpass  ①
import smtplib  ②
from email.message import EmailMessage  ③
from datetime import datetime

TIMESTAMP = datetime.now().ctime()  ④

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:")  ⑤

smtpserver = smtplib.SMTP("smtp2go.com", 2525)  ⑥
smtpserver.login(SMTP_USER, SMTP_PASSWORD)  ⑦

msg = EmailMessage()  ⑧
msg.set_content(MESSAGE_BODY)  ⑨
msg['Subject'] = MESSAGE_SUBJECT  ⑩
msg['from'] = SENDER  ⑪
msg['to'] = RECIPIENTS  ⑫

try:
    smtpserver.send_message(msg)  ⑬
except smtplib.SMTPException as err:
    print("Unable to send mail:", err)
finally:
    smtpserver.quit()  ⑭
```

① module for hiding password

② module for sending email

③ module for creating message

④ get a time string for the current date/time

⑤ get password (not echoed to screen)

⑥ connect to SMTP server

⑦ log into SMTP server

⑧ create empty email message

⑨ add the message body

⑩ add the message subject

⑪ add the sender address

⑫ add a list of recipients

⑬ send the message

⑭ 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

```
#!/usr/bin/env python
import smtplib
from datetime import datetime
from imghdr import what  ①
from email.message import EmailMessage ②
from getpass import getpass ③


SMTP_SERVER = "smtp2go.com"  ④
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()  ⑤
    msg.set_content(body)  ⑥
    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:  ⑦
        attachment_data = file_in.read()
    message.add_attachment(attachment_data)  ⑧


def add_image_attachment(file_name, message):
    with open(file_name, 'rb') as file_in:  ⑨
        attachment_data = file_in.read()
    image_type = what(None, h=attachment_data)  ⑩
    message.add_attachment(attachment_data, maintype='image', subtype=image_type)
⑪


def create_smtp_server():
    password = getpass("Enter SMTP server password:")  ⑫
    smtpserver = smtplib.SMTP(SMTP_SERVER, SMTP_PORT)  ⑬
    smtpserver.login(SMTP_USER, password)  ⑭

    return smtpserver


def send_message(server, message):
    try:
        server.send_message(message)  ⑮
    finally:
        server.quit()


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

① module to determine image type

② module for creating email message

③ module for reading password privately

④ global variables for external information (IRL should be from environment — command line, config file, etc.)

⑤ create instance of EmailMessage to hold message

⑥ set content (message text) and various headers

⑦ read data for text attachment

⑧ add text attachment to message

⑨ read data for binary attachment

⑩ get type of binary data

⑪ add binary attachment to message, including type and subtype (e.g., "image/jpg)"

⑫ get password from user (don't hardcode sensitive data in script)

⑬ create SMTP server connection

⑭ log into SMTP connection

⑮ send message

Remote Access

- Use paramiko (not part of standard library)
- Create ssh client
- Create transport object to use sftp

For remote access to other computers, you would usually use the ssh protocol. Python has several ways to use ssh.

The 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 Anaconda.

To run commands on a remote computer, use SSHClient. Once you connect to the remote host, you can execute commands and retrieve the standard I/O of the remote program.

To avoid the "I haven't seen this host before" prompt, call **set_missing_host_key_policy()** like this:

```
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
```

The exec_command() method executes a command on the remote host, and returns a triple with the remote command's stdin, stdout, and stderr as file-like objects.

There is also a builtin **ssh** module, but it depends on having an external command named "ssh".

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/

Example

paramiko_commands.py

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

import paramiko

with paramiko.SSHClient() as ssh:  ①

    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())  ②

    ssh.connect('localhost', username='python', password='l0lz')  ③

    stdin, stdout, stderr = ssh.exec_command('whoami')  ④
    print(stdout.read().decode())  ⑤
    print('-' * 60)

    stdin, stdout, stderr = ssh.exec_command('ls -l')  ④
    print(stdout.read().decode())  ⑤
    print('-' * 60)

    stdin, stdout, stderr = ssh.exec_command('ls -l /etc/passwd /etc/horcrux')  ④
    print("STDOUT:")
    print(stdout.read().decode())  ⑤
    print("STDERR:")
    print(stderr.read().decode())  ⑥
    print('-' * 60)
```

① create paramiko client

② ignore missing keys (this is safe)

③ connect to remote host

④ execute remote command; returns standard I/O objects

⑤ read stdout of command

⑥ read stderr of command

paramiko_commands.py

```
python

------------------------------------------------------------
total 8
drwx------+  3 python  staff   96 Apr 16  2014 Desktop
drwx------+  3 python  staff   96 Apr 16  2014 Documents
drwx------+  4 python  staff  128 Apr 16  2014 Downloads
drwx------+ 28 python  staff  896 Dec 17  2017 Library
drwx------+  3 python  staff   96 Apr 16  2014 Movies
drwx------+  3 python  staff   96 Apr 16  2014 Music
drwx------+  3 python  staff   96 Apr 16  2014 Pictures
drwxr-xr-x+  5 python  staff  160 Apr 16  2014 Public
-rw-r--r--   1 python  staff  519 Jul 27  2016 remote_processes.py
drwxr-xr-x   2 python  staff   64 Jan  1 17:55 text_files

------------------------------------------------------------
STDOUT:
-rw-r--r--  1 root  wheel  6946 Oct 17 18:39 /etc/passwd

STDERR:
ls: /etc/horcrux: No such file or directory

------------------------------------------------------------
```

Copying files with Paramiko

- 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

```
#!/usr/bin/env python
import os
import paramiko

REMOTE_DIR = 'text_files'

with paramiko.Transport(('localhost', 22)) as transport:  ①
    transport.connect(username='python', password='l0lz')  ②
    sftp = paramiko.SFTPClient.from_transport(transport)  ③
    for item in sftp.listdir_iter():  ④
        print(item)
    print('-' * 60)

    remote_file = os.path.join(REMOTE_DIR, 'betsy.txt')  ⑤

    sftp.mkdir(REMOTE_DIR)  ⑥
    sftp.put('../DATA/alice.txt', remote_file)  ⑦
    sftp.get(remote_file, 'eileen.txt')  ⑧

with paramiko.SSHClient() as ssh:  ⑨
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

    try:
        ssh.connect('localhost', username='python', password='l0lz')
    except paramiko.SSHException as err:
        print(err)
        exit()

    stdin, stdout, stderr = ssh.exec_command('ls -l {}'.format(REMOTE_DIR))
    print(stdout.read().decode())
    print('-' * 60)

    stdin, stdout, stderr = ssh.exec_command('rm -f
{}/betsy.txt'.format(REMOTE_DIR))
    stdin, stdout, stderr = ssh.exec_command('rmdir {}'.format(REMOTE_DIR))
    stdin, stdout, stderr = ssh.exec_command('ls -l')
    print(stdout.read().decode())
    print('-' * 60)
```

① create paramiko Transport instance

② connect to remote host

③ create SFTP client using Transport instance

④ get list of items on default (login) folder (listdir_iter() returns a generator)

⑤ create path for remote file

⑥ create a folder on the remote host

⑦ copy a file to the remote host

⑧ copy a file from the remote host

⑨ use SSHClient to confirm operations (not needed, just for illustration)

paramiko_copy_files.py

```
drwx------   1 504      20             96 16 Apr 2014  Music
-rw-------   1 504      20              3 16 Apr 2014  .CFUserTextEncoding
drwx------   1 504      20             96 16 Apr 2014  Pictures
drwxr-xr-x   1 504      20             64 01 Jan 17:55 text_files
drwx------   1 504      20             96 16 Apr 2014  Desktop
drwx------   1 504      20            896 17 Dec 2017  Library
drwxr-xr-x   1 504      20            160 16 Apr 2014  Public
drwx------   1 504      20             96 06 Feb 2015  .ssh
drwx------   1 504      20             96 16 Apr 2014  Movies
drwx------   1 504      20             96 16 Apr 2014  Documents
-rw-r--r--   1 504      20            519 27 Jul 2016  remote_processes.py
drwx------   1 504      20            128 16 Apr 2014  Downloads
-rw-------   1 504      20           2614 23 May 2019  .bash_history
------------------------------------------------------------
```

Chapter 12 Exercises

Exercise 12-1 (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 12-2 (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". (For *real* screen-scraping, you can use the Beautiful Soup module.)

You can use the string method **find()**, which can be called like S.find('text', start, stop), which finds on a slice of the string, moving forward each time the string is found.

Exercise 12-3 (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 A: Python Bibliography

Title	Author	Publisher
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

Title	Author	Publisher
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, 3nd. 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

Title	Author	Publisher
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

Title	Author	Publisher
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

Index

