Extra Topics for NNL

John Strickler

Version 1.0, January 2021

Table of Contents

Chapter 1: 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

```
// or maybe one of the standard datasets, like irises
```

Python’s scientific stack

- NumPy, SciPy, MatPlotLib (and many others)
- Python extensions, written in C/Fortran
- Support for math, numerical, and scientific operations

NumPy is part of what is sometimes called Python’s "scientific stack". Along with SciPy, Matplotlib, and other libraries, it provides a broad range of support for scientific and engineering tasks.

SciPy is a large group of mathematical algorithms, along with some convenience functions, for doing scientific and engineering calculations, including data science. SciPy routines accept and return NumPy arrays.

pandas ties some of the libraries together, and is frequently used interactively via **iPython** in a **Jupyter** notebook. Of course you can also create scripts using any of the scientific libraries.

NOTE There is not an integrated *application* for all of the Python scientific libraries.

NumPy overview

- Install numpy module from numpy.scipy.org (included with Anaconda)
- Basic object is the array
- Up to 100x faster than normal Python math operations
- Functional-based (fewer loops)
- Dozens of utility functions

The basic object that NumPy provides is the array. Arrays can have as many dimensions as needed. Working with NumPy arrays can be 100 times faster than working with normal Python lists.

Operations are applied to arrays in a functional manner – instead of the programmer explicitly looping through elements of the array, the programmer specifies an expression or function to be applied, and the array object does all of the iteration internally.

There are many utility functions for accessing arrays, for creating arrays with specified values, and for performing standard numerical operations on arrays.

To get started, import the **numpy** module. It is conventional to import numpy as np. The examples in this chapter will follow that convention.

NumPy and the rest of the Python scientific stack is included with the Anaconda, Canopy, Python(x,y), and WinPython bundles. If you are not using one of these, install NumPy with

```
pip install numpy
```

NOTE all top-level NumPy routines are also available directly through the scipy package.

Creating Arrays

- Create with
  - array() function initialized with nested sequences
  - Other utilities (arange(), zeros(), ones(), empty()
- All elements are same type (default float)
- Useful properties: ndim, shape, size, dtype
- Can have any number of axes (dimensions)
- Each axis has a length

An array is the most basic object in NumPy. It is a table of numbers, indexed by positive integers. All of the elements of an array are of the same type.

An array can have any number of dimensions; these are referred to as axes. The number of axes is called the rank.

Arrays are rectangular, not ragged.

One way to create an array is with the `array()` function, which can be initialized from existing arrays.

The `zeros()` function expects a *shape* (tuple of axis lengths), and creates the corresponding array, with all values set to zero. The `ones()` function is the same, but initializes with ones.

The `full()` function expects a shape and a value. It creates the array, putting the specified value in every element.

The `empty()` function creates an array of specified shape initialized with random floats.

However, the most common way to crate an array is by loading data from a text or binary file.

When you print an array, NumPy displays it with the following layout:

- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

NOTE the `ndarray()` object is initialized with the *shape*, not the *data*.

Example

np_create_arrays.py

```
import 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.]]]]

[[ 2.00000000e+000  2.00000000e+000  9.70370850e-001  9.70370850e-001
   1.24550326e-001  4.68729519e-001  7.13074033e-001  4.91525424e-001]
 [-1.05842149e-001  2.52857143e+001 -2.94438241e+001  6.01019542e+000
  -2.94438241e+001  3.72857143e+001 -9.44803187e+000  6.01019542e+000]
```

```
 [-9.44803187e+000  4.42857143e+000  6.36598737e-314  6.95208261e-310
   2.00000000e+000 -4.34516446e-311  6.95226831e-310  1.73832808e-308]]
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()`, or Python slices in general. Unlike the builtin Python `range()`, start, stop, and step can be floats.

The `linspace()` function creates a specified number of equally-spaced values. As with `numpy.arange()`, start and stop may be floats.

The resulting arrays can be reshaped into multidimensional arrays.

Example

np_create_ranges.py

```
#!/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.

In-place operators (+=, *=, etc) efficiently modify the array itself, rather than returning a new array.

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]]
```

Shapes

- Number of elements on each axis
- array.shape has shape tuple
- Assign to array.shape to change
- Convert to one dimension
  - array.ravel()
  - array.flatten()
- array.transpose() to flip the shape

Every array has a shape, which is the number of elements on each axis. For instance, an array might have the shape (3,5), which means that there are 3 rows and 5 columns.

The shape is stored as a tuple, in the shape attribute of an array. To change the shape of an array, assign to the shape attribute.

The `ravel()` and `flatten()` methods will flatten any array into a single dimension. `ravel()` returns a "view" of the original array, while `flatten()` returns a new array. If you modify the result of ravel(), it will modify the original data.

The `transpose()` method will flip shape (x,y) to shape (y,x). It is equivalent to `array.shape = list(reversed(array.shape))`.

Example

np_shapes.py

```
#!/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]]
```

Selecting rows based on conditions

- Index with boolean expressions
- Use **&**, not **and**

To select rows from an array, based on conditions, you can index the array with two or more Boolean expressions.

Since the Boolean expressions return arrays of True/False values, use the **&** bitwise AND operator (or **|** for OR).

Any number of conditions can be applied this way.

```
new_array = old_array[bool_expr1 & bool_expr2 ...]
```

Example

np_select_rows.py

```
#!/usr/bin/env python
import numpy as np

sample_data = np.loadtxt(   ①
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
)

print("first 5 rows of sample_data:")
print(sample_data[:5, :], '\n')

selected = sample_data[  ②
    (sample_data[:, 0] < 10) &  ③
    (sample_data[:, -1] > 35)
]

print("selected")
print(selected)
```

① Read some data into 2d array

② Index into the existing data

③ Combine two Boolean expressions with **&**

np_select_rows.py

```
first 5 rows of sample_data:
[[63. 51. 59. 61. 50.  4.]
 [40. 66.  9. 64. 63. 17.]
 [18. 23.  2. 61.  1.  9.]
 [29.  8. 40. 59. 10. 26.]
 [54.  9. 68.  4. 16. 21.]]

selected
[[ 8. 49.  2. 40. 50. 36.]
 [ 4. 49. 39. 50. 23. 39.]
 [ 6.  7. 40. 56. 31. 38.]
 [ 6.  1. 44. 55. 49. 36.]
 [ 5. 22. 45. 49. 10. 37.]]
```

Stacking

- Combining 2 arrays vertically or horizontally
- use `vstack()` or `hstack()`
- Arrays must have compatible shapes

You can combine two or more arrays vertically or horizontally with the `vstack()` or `hstack()` functions. These functions are also handy for adding rows or columns with the results of operations.

Example

np_stacking.py

```
#!/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]]
```

ufuncs and builtin operators

- Builtin functions for efficiency
- Map over array
- No **for** loops
- Use **vectorize()** for custom ufuncs

In normal Python, you are used to iterating over arrays, especially nested arrays, with a **for** loop. However, for large amounts of data, this is slow. The reason is that the interpreter must do type-checking and lookups for each item being looped over.

NumPy provides *vectorized* operations which are implemented by *ufuncs* — universal functions. ufuncs are implemented in C and work directly on NumPy arrays. When you use a normal math operator (**+ - * /**, etc) on a NumPy array, it calls the underlying ufunc. For instance, `array1 + array2` calls `np.add(array1, array2)`.

There are over 60 ufuncs built into NumPy. These normally return a NumPy array with the results of the operation. Some have options for putting the output into a different object.

The official docs for ufuncs are here: https://docs.scipy.org/doc/numpy/reference/ufuncs.html

You can scroll down to the list of available ufuncs.

Vectorizing functions

- Many functions "just work"
- `np.vectorize()` allows user-defined function to be broadcast.

ufuncs will automatically be broadcast across any array to which they are applied. For user-defined functions that don’t correctly broadcast, NumPy provides the **vectorize()** function. It takes a function which accepts one or more scalar values (float, integers, etc.) and returns a single scalar value.

Example

np_vectorize.py

```
#!/usr/bin/env python
import time
import numpy as np

sample_data = np.loadtxt(   ①
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
)

def set_default(value, limit, default): ②
    if value > limit:
        value = default

    return value

MAX_VALUE = 50      ③
DEFAULT_VALUE = -1  ④

print("Version 1: looping over arrays")
start = time.time() ⑤
try:
    version1_array = np.zeros(sample_data.shape, dtype=int)  ⑥
    for i, row in enumerate(sample_data):  ⑦
        for j, column in enumerate(row):
            version1_array[i, j] = set_default(sample_data[i, j], MAX_VALUE,
DEFAULT_VALUE)   ⑧
except ValueError as err:
    print("Function failed:", err)
else:
    end = time.time()  ⑨
    elapsed = end - start  ⑩
    print(version1_array)
    print("took {:.5f} seconds".format(elapsed))
finally:
    print()


print("Version 2: broadcast without vectorize()")
start = time.time()
try:
    print("Without sp.vectorize:")
    version2_array = set_default(sample_data, MAX_VALUE, DEFAULT_VALUE) ⑪
except ValueError as err:
    print("Function failed:", err)
```



```
else:
    end = time.time()
    elapsed = end - start
    print(version2_array)
    print("took {:.5f} seconds".format(elapsed))
finally:
    print()

print("Version 3: broadcast with vectorize()")
set_default_vect = np.vectorize(set_default)  ⑫

start = time.time()
try:
    print("With sp.vectorize:")
    version3_array = set_default_vect(sample_data, MAX_VALUE, DEFAULT_VALUE) ⑬
except ValueError as err:
    print("Function failed:", err)
else:
    end = time.time()
    elapsed = end - start
    print(version3_array)
    print("took {:.5f} seconds".format(elapsed))
finally:
    print()
```

① Create some sample data

② Define function with more than one parameter

③ Define max value

④ Define default value

⑤ Get the current time as Unix timestamp (large integer)

⑥ Create array to hold results

⑦ Iterate over rows and columns of input array

⑧ Call function and put result in new array

⑨ Get current time

⑩ Get elapsed number of seconds and print them out

⑪ Pass array to function; it fails because it has more than one parameter

⑫ Convert function to vectorized version — creates function that takes one parameter and has the other two "embedded" in it

⑬ Call vectorized version with same parameters *np_vectorize.py*

```
Version 1: looping over arrays
[[-1 -1 -1 -1 50  4]
 [40 -1  9 -1 -1 17]
 [18 23  2 -1  1  9]
 ...
 [26 20 -1 46 38 23]
 [ 9  5 -1 23  2 26]
 [46 34 25  8 39 34]]
took 0.00460 seconds

Version 2: broadcast without vectorize()
Without sp.vectorize:
Function failed: The truth value of an array with more than one element is ambiguous. Use
a.any() or a.all()

Version 3: broadcast with vectorize()
With sp.vectorize:
[[-1 -1 -1 -1 50  4]
 [40 -1  9 -1 -1 17]
 [18 23  2 -1  1  9]
 ...
 [26 20 -1 46 38 23]
 [ 9  5 -1 23  2 26]
 [46 34 25  8 39 34]]
took 0.00075 seconds
```

Getting help

- Several help functions
  - `numpy.info()`
  - `numpy.lookfor()`
  - `numpy.source()`

NumPy has several functions for getting help. The first is `numpy.info()`, which provides a brief explanation of a function, class, module, or other object as well as some code examples.

If you're not sure what function you need, you can try `numpy.lookfor()`, which does a keyword search through the NumPy documentation.

These functions are convenient when using **iPython** or **Jupyter**.

Example

np_info.py

```
#!/usr/bin/env python
import numpy as np
import scipy.fftpack as ff


def main():
    np.info(ff.fft)  ①

    print('-' * 60)

    np.source(ff.fft)  ②




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

① Get help on the fft() function

② View the source of the fft() function

Iterating

- Similar to normal Python
- Iterates through first dimension
- Use array.flat to iterate through all elements
- Don't do it unless you have to

Iterating through a NumPy array is similar to iterating through any Python list; iteration is across the first dimension. Slicing and indexing can be used.

To iterate across every element, use `array.flat`.

However, iterating over a NumPy array is generally much less efficient than using a *vectorized* approach — calling a *ufunc* or directly applying a math operator. Some tasks may require it, but you should avoid it if possible.

Example

np_iterating.py

```
#!/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
```

Matrix Multiplication

- Use normal ndarrays
- Most operations same as ndarray
- Use @ for multiplication

For traditional matrix operations, use a normal ndarray. Most operations are the same as for ndarrays. For matrix (diagonal) multiplication, use the @ (matrix multiplication) operator.

For transposing, use *array*.transpose(), or just *array*.T.

NOTE There was formerly a Matrix type in NumPy, but it is deprecated since the addition of the @ operator in Python 3.5

Example

np_matrices.py

```
#!/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

- Default is **float**
- Data type is inferred from initialization data
- Can be specified with `arange()`, `ones()`, `zeros()`, etc.

Numpy defines around 30 numeric data types. Integers can have different sizes and byte orders, and be either signed or unsigned. The data type is normally inferred from the initialization data. When using `arange()`, `ones()`, etc., to create arrays, the **dtype** parameter can be used to specify the data type.

The default data type is **np.float_**, which maps to the Python builtin type **float**.

The data type cannot be changed after an array is created.

See https://numpy.org/devdocs/user/basics.types.html for more details.

Example

np_data_types.py

```
#!/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]]]
```

Reading and writing Data

- Read data from files into **ndarray**
- Text files
  - `loadtxt()`
  - `savetxt()`
  - `genfromtxt()`
- Binary (or text) files
  - `fromfile()`
  - `tofile()`

NumPy has several functions for reading data into an array.

`numpy.loadtxt()` reads a delimited text file. There are many options for fine-tuning the import.

`numpy.genfromtxt()` is similar to `numpy.loadtxt()`, but also adds support for handling missing data

Both functions allow skipping rows, user-defined per-column converters, setting the data type, and many others.

To save an array as a text file, use the `numpy.savetxt()` function. You can specify delimiters, header, footer, and formatting.

To read binary data, use `numpy.fromfile()`. It expects a file to contain all the same data type, i.e., ints or floats of a specified type. It will default to floats. `fromfile()` can also be used to read text files.

To save as binary data, you can use `numpy.tofile()`, but **tofile()** and **fromfile()** are not platform-independent. See the next section on **save()** and **load()** for platform-independent I/O.

Example

np_savetxt_loadtxt.py

```
#!/usr/bin/env python
import numpy as np

sample_data = np.loadtxt(   ①
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
    dtype=float
)

print(sample_data)
print('-' * 60)

sample_data  /= 10  ②

float_file_name = 'save_data_float.txt'

np.savetxt(float_file_name, sample_data, delimiter=",", fmt="%5.2f")  ③

int_file_name = 'save_data_int.txt'

np.savetxt(int_file_name, sample_data, delimiter=",", fmt="%d")  ④

data = np.loadtxt(float_file_name, delimiter=",")  ⑤
print(data)
```

① Load data from space-delimited file

② Modify sample data

③ Write data to text file as floats, rounded to two decimal places, using commas as delimiter

④ Write data to text file as ints, using commas as delimiter

⑤ Read data back into **ndarray**

np_savetxt_loadtxt.py

```
[[63. 51. 59. 61. 50.  4.]
 [40. 66.  9. 64. 63. 17.]
 [18. 23.  2. 61.  1.  9.]
 ...
 [26. 20. 54. 46. 38. 23.]
 [ 9.  5. 59. 23.  2. 26.]
 [46. 34. 25.  8. 39. 34.]]
------------------------------------------------------------
[[6.3 5.1 5.9 6.1 5.  0.4]
 [4.  6.6 0.9 6.4 6.3 1.7]
 [1.8 2.3 0.2 6.1 0.1 0.9]
 ...
 [2.6 2.  5.4 4.6 3.8 2.3]
 [0.9 0.5 5.9 2.3 0.2 2.6]
 [4.6 3.4 2.5 0.8 3.9 3.4]]
```


Example

np_tofile_fromfile.py

```
#!/usr/bin/env python
import numpy as np

sample_data = np.loadtxt(   ①
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
    dtype=float
)

sample_data  /= 10  ②

print(sample_data)
print("-" * 60)

file_name = 'sample.dat'

sample_data.tofile(file_name)  ③

data = np.fromfile(file_name)  ④
data.shape = sample_data.shape  ⑤

print(data)
```

① Read in sample data

② Modify sample data

③ Write data to file (binary, but not portable)

④ Read binary data from file as one-dimensional array

⑤ Set shape to shape of original array

np_tofile_fromfile.py

```
[[6.3 5.1 5.9 6.1 5.  0.4]
 [4.  6.6 0.9 6.4 6.3 1.7]
 [1.8 2.3 0.2 6.1 0.1 0.9]
 ...
 [2.6 2.  5.4 4.6 3.8 2.3]
 [0.9 0.5 5.9 2.3 0.2 2.6]
 [4.6 3.4 2.5 0.8 3.9 3.4]]
------------------------------------------------------------
[[6.3 5.1 5.9 6.1 5.  0.4]
 [4.  6.6 0.9 6.4 6.3 1.7]
 [1.8 2.3 0.2 6.1 0.1 0.9]
 ...
 [2.6 2.  5.4 4.6 3.8 2.3]
 [0.9 0.5 5.9 2.3 0.2 2.6]
 [4.6 3.4 2.5 0.8 3.9 3.4]]
```


Saving and retrieving arrays

- Efficient binary format
- Save as NumPy data
  - Use `numpy.save()`
- Read into ndarray
  - Use `numpy.load()`

To save an array as a NumPy data file, use `numpy.save()`. This will write the data out to a specified file name, adding the extension '.npy'.

To read the data back into a NumPy ndarray, use `numpy.load()`. Data are read and written in a way that preserves precision and endianness.

This the most efficient way to store numeric data for later retrieval, compared to **savetext()** and **loadtext()** or **tofile()** and **fromfile()**. Files written with `numpy.save()` are not human-readable.

Example

np_save_load.py

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


import numpy as np

sample_data = np.loadtxt(   ①
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
    dtype=int
)

sample_data  *= 100  ②

print(sample_data)

file_name = 'sampledata'

np.save(file_name, sample_data)  ③

retrieved_data = np.load(file_name + '.npy')  ④

print('-' * 60)
print(retrieved_data)
```

① Read some sample data into an ndarray

② Modify the sample data (multiply every element by 100)

③ Write entire array out to NumPy-format data file (adds **.npy** extension)

④ Retrieve data from saved file

np_save_load.py

```
[[6300 5100 5900 6100 5000  400]
 [4000 6600  900 6400 6300 1700]
 [1800 2300  200 6100  100  900]
 ...
 [2600 2000 5400 4600 3800 2300]
 [ 900  500 5900 2300  200 2600]
 [4600 3400 2500  800 3900 3400]]
------------------------------------------------------------
[[6300 5100 5900 6100 5000  400]
 [4000 6600  900 6400 6300 1700]
 [1800 2300  200 6100  100  900]
 ...
 [2600 2000 5400 4600 3800 2300]
 [ 900  500 5900 2300  200 2600]
 [4600 3400 2500  800 3900 3400]]
```

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. (The **c_** object is the same, but is column-oriented).

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 that 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.

TIP If the first element in the index 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.
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.matrix'>
```


Chapter 1 Exercises

Exercise 1-1 (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 1-2 (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 1-3 (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 2: Introduction to pandas

Objectives

- Understand what the pandas module provides
- Load data from CSV and other files
- Access data tables
- Extract rows and columns using conditions
- Calculate statistics for rows or columns

About pandas

- Reads data from file, database, or other sources
- Deals with real-life issues such as invalid data
- Powerful selecting and indexing tools
- Builtin statistical functions
- Munge, clean, analyze, and model data
- Works with numpy and matplotlib

pandas is a package designed to make it easy to get, organize, and analyze large datasets. Its strengths lie in its ability to read from many different data sources, and to deal with real-life issues, such as missing, incomplete, or invalid data.

pandas also contains functions for calculating means, sums and other kinds of analysis.

For selecting desired data, pandas has many ways to select and filter rows and columns.

It is easy to integrate pandas with NumPy, Matplotlib, and other scientific packages.

While pandas can handle three (or higher) dimensional data, it is generally used with two-dimensional (row/column) data, which can be visualized like a spreadsheet.

pandas provides powerful split-apply-combine operations — **groupby** enables transformations, aggregations, and easy-access to plotting functions. It is easy to emulate R's plyr package via pandas.

NOTE pandas gets its name from *pan*el *da*ta *s*ystem

pandas architecture

- Two main structures: Series and DataFrame
- Series – one-dimensional
- DataFrame – two-dimensional

The two main data structures in pandas are the **Series** and the **DataFrame**. A series is a one-dimensional indexed list of values, something like an ordered dictionary. A DataFrame is is a two-dimensional grid, with both row and column 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.

NOTE pandas used to support the **Panel** type, which is more more or less a collection of DataFrames, but Panel has been deprecated in favor of hierarchical indexing.

Series

- Indexed list of values
- Similar to a dictionary, but ordered
- Can get sum(), mean(), etc.
- Use index to get individual values
- 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 1. DataFrame Initializers

Initializer	Description
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the “dict of arrays” case
dict of Series	Each value becomes a column. Indexes from each Series are union-ed together to form the result’s row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column. Keys are union-ed to form the row index as in the “dict of Series” case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame’s column labels
List of lists or tuples	Treated as the “2D ndarray” case
Another DataFrame	The DataFrame’s indexes are used unless different ones are passed
NumPy MaskedArray	Like the “2D ndarray” case except masked values become NA/missing in the DataFrame result

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
```

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
```

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
```

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 2. 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    0.492520
2013-04-01 10:01:00    0.238449
2013-04-01 10:02:00    0.358703
2013-04-01 10:03:00    0.566284
2013-04-01 10:04:00    0.594679
2013-04-01 10:05:00    0.674658
2013-04-01 10:06:00    0.857848
2013-04-01 10:07:00    0.438616
2013-04-01 10:08:00    0.578420
2013-04-01 10:09:00    0.724035
2013-04-01 10:10:00    0.015520
2013-04-01 10:11:00    0.051806
2013-04-01 10:12:00    0.280377
2013-04-01 10:13:00    0.148973
2013-04-01 10:14:00    0.888725
2013-04-01 10:15:00    0.689131
2013-04-01 10:16:00    0.922678
2013-04-01 10:17:00    0.252500
2013-04-01 10:18:00    0.933474
2013-04-01 10:19:00    0.419521
2013-04-01 10:20:00    0.202447
2013-04-01 10:21:00    0.146945
2013-04-01 10:22:00    0.692803
2013-04-01 10:23:00    0.503386
2013-04-01 10:24:00    0.067124
2013-04-01 10:25:00    0.544836
2013-04-01 10:26:00    0.318833
2013-04-01 10:27:00    0.637881
2013-04-01 10:28:00    0.115496
2013-04-01 10:29:00    0.390126
2013-04-01 10:30:00    0.446892
Freq: T, dtype: float64
```

Useful pandas methods

Table 3. Methods and attributes for fetching DataFrame/Series data

Method	Description
DF.columns()	Get or set column labels
DF.shape() S.shape()	Get or set shape (length of each axis)
DF.head(n) DF.tail(n)	Return n items (default 5) from beginning or end
DF.describe() S.describe()	Display statistics for dataframe
DF.info()	Display column attributes
DF.values S.values	Get the actual values from a data structure
DF.loc[row_indexer[1], col_indexer]	Multi-axis indexing by label (not by position)
DF.iloc[row_indexer[2], col_indexer]	Multi-axis indexing by position (not by labels)

[1] Indexers can be label, slice of labels, or iterable of labels.
[2] Indexers can be numeric index (0-based), slice of indexes, or iterable of indexes.

Table 4. Methods for Computations/Descriptive Stats (called from pandas)

Method	Returns
abs()	absolute values
corr()	pairwise correlations
count()	number of values
cov()	Pairwise covariance
cumsum()	cumulative sums
cumprod()	cumulative products
cummin(), cummax()	cumulative minimum, maximum
kurt()	unbiased kurtosis
median()	median
min(), max()	minimum, maximum values
prod()	products
quantile()	values at given quantile
skew()	unbiased skewness
std()	standard deviation
var()	variance

NOTE these methods return Series or DataFrames, as appropriate, and can be computed over rows (axis=0) or columns (axis=1). They generally skip NA/null values.

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_… functions have many options for controlling and parsing input. For instance, if large integers in the file contain commas, the thousands options let you set the separator as comma (in the US), so it will ignore them.

read_csv() is the most frequently used function, and has many options. It can also be used to read generic flat-file formats. **read_table** is similar to **read_csv()**, but doesn't assume CSV format.

There are corresponding "to_…." functions for many of the read functions. `to_csv()` and `to_ndarray()` are particular useful.

These are not the only functions for reading into

See https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html?highlight=output#io-html for details on the I/O functions.

NOTE See Jupyter notebook **pandas_Input_Demo** for examples of reading most types of input.

Table 5. pandas I/O functions

Format	Input function	Output function
CSV	read_csv()	to_csv()
Delimited file (generic)	read_table()	to_csv()
Excel worksheet	read_excel()	to_excel()
File with fixed-width fields	read_fwf()	
Google BigQuery	read_gbq()	to_gbq()
HDF5	read_hdf()	to_hdf()
HTML table	read_html()	to_html()
JSON	read_json()	to_json()
OS clipboard data	read_clipboard()	to_clipboard()
Parquet	read_parquet()	to_parquet()
pickle	read_pickle()	to_pickle()
SAS	read_sas()	
SQL query	read_sql()	to_sql()

NOTE All **read_...()** functions return a new **DataFrame**, except **read_html()**, which returns a list of **DataFrames**

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 2 Exercises

Exercise 2-1 (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 2-2 (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 3: 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 3 Exercises

Exercise 3-1 (energy_use_plot.py)

Using the file energy_use_quad.csv in the DATA folder, use matplotlib to plot the data for "Transportation", "Industrial", and "Residential and Commercial". Don't plot the "as a percent…".

You can do this in iPython, or as a standalone script. If you create a standalone script, save the figure to a file, so you can view it.

Use pandas to read the data. The columns are, in Python terms:

```
['Desc',"1960","1965","1970","1975","1980","1985","1990","1991","1992","1993","1994","199
5","1996","1997","1998","1999","2000","2001","2002","2003","2004","2005","2006","2007","2
008","2009","2010","2011"]
```

TIP See the script pandas_energy.py in the EXAMPLES folder to see how to load the data.

Chapter 4: Database Access

Objectives

- Understand the Python DB API architecture
- Connect to a database
- Execute simple and parameterized queries
- Fetch single and multiple row results
- Get metadata about a query
- Execute non-query statements
- Start transactions and commit or rollback as needed

The DB API

- Several ways to access DBMSs from Python
- DB API is most popular
- DB API is sort of an "abstract class"
- Many modules for different DBMSs using DB API
- Hides actual DBMS implementation

To make database programming simpler, Python has the DB API. This is an API to standardize working with databases. When a package is written to access a database, it is written to conform to the API, and thus programmers do not have to learn a new set of methods and functions.

Table 6. Available Interfaces (using Python DB API-2.0)

Database	Python package
Firebird (and Interbase)	KInterbasDB
IBM DB2	PyDB2
Informix	informixdb
Ingres	ingmod
Microsoft SQL Server	pymssql
MySQL	pymysql
ODBC	pyodbc
Oracle	cx_oracle
PostgreSQL	psycopg2
SAP DB (also known as "MaxDB")	sapdbapi
SQLite	sqlite3
Sybase	Sybase

NOTE There may be other interfaces to some of the listed DBMSs as well.

Connecting to a Server

- Import appropriate library
- Use connect() to get a database object
- Specify host, database, username, password

To connect to a database server, import the package for the specific database. Use the package's **connect()** method to get a database object, specifying the host, initial database, username, and password. If the username and password are not needed, use None.

Some database modules have nonstandard parameters to the connect() method.

When finished with the connection, call the **close()** method on the connection object.

Many database modules support the context manager (**with** statement), and will automatically close the database when the with block is exited. Check the documentation to see how this is implemented for a specific database.

Example

```
import sqlite3

slconn = sqlite3.connect('web_content')

import pymysql

myconn = pymysql.connect (host = "myserver1",
                          user = "adeveloper",
                          passwd = "s3cr3t",
                          db = "web_content")
#  make queries, etc. here ...
myconn.close()
```

NOTE Argument names for the connect() method may not be consistent. For instance, pymysql supports the above parameter names, while pymssql does not.

Table 7. connect() examples

Package	Database	Connection
cx_oracle	Oracle	`ip = 'localhost'` `port = 1521` `SID = 'YOURSIDHERE'` `dsn_tns = cx_Oracle.makedsn(ip, port, SID)` `db = cx_Oracle.connect('adeveloper', '$3cr3t', dsn_tns)`
psychopg	PostgreSQL	`psycopg2.connect ('''` `host='localhost'` `user='adeveloper'` `password='$3cr3t'` `dbname='testdb'` `''')` *note: connect() has one (string) parameter, not multiple parameters*
pymssql	MS-SQL	`pymssql.connect (` `host="localhost",` `user="adeveloper",` `passwd="$3cr3t",` `db="testdb",` `)` `pymssql.connect (` `dsn="DSN",` `)`
pymysql	MySQL	`pymysql.connect (` `host="localhost",` `user="adeveloper",` `passwd="$3cr3t",` `db="testdb",` `)`
pyodbc	Any ODBC-compliant DB	`pyodbc.connect('''` `DRIVER={SQL Server};` `SERVER=localhost;` `DATABASE=testdb;` `UID=adeveloper;` `PWD=$3cr3t` `''')` `pyodbc.connect('DSN=testdsn;PWD=$3cr3t')` *note: connect() has one (string) parameter, not multiple parameters*
sqlite3	SqlLite3	`sqlite3.connect('testdb')` `sqlite3.connect(':memory:')`

Creating a Cursor

- Cursor can execute SQL statements
- Multiple cursors available
  - Standard cursor
    - Returns tuples
  - Other cursors
    - Returns dictionaries
    - Leaves data on server

Once you have a database object, you can create one or more cursors. A cursor is an object that can execute SQL code and fetch results.

The default cursor for most packages returns each row as a tuple of values. There are different types of cursors that can return data in different formats, or that control whether data is stored on the client or the server.

Example

```
myconn = pymysqlconnect (host="myserver1",user="adeveloper", passwd="s3cr3t",
db="web_content")
mycursor = myconn.cursor()
```

Executing a Statement

- Executing cursor sends SQL to server
- Data not returned until asked for
- Returns number of lines in result set for queries
- Returns lines affected for other statements

Once you have a cursor, you can use it to perform queries, or to execute arbitrary SQL statements via the execute() method. The first argument to execute() is a string containing the SQL statement to run.

Example

```
cursor.execute("select hostname,ostype,user from hostinfo")
cursor.execute('insert into hostinfo values
("foo",5,"2.6","arch","net",2055,3072,"bob",0)')
```


Fetching Data

- Use one of the fetch methods from the cursor object
- Syntax
  - `rec = cursor.fetchone()`
  - `recs = cursor.fetchall()`
  - `recs = cursor.fetchmany()`

Cursors provide three methods for returning query results.

fetchone() returns the next available row from the query results.

fetchall() returns a tuple of all rows.

fetchmany(n) returns up to n rows. This is useful when the query returns a large number of rows.

Example

```
cursor.execute("select color, quest from knights where name = 'Robin'")
(color,quest) = cursor.fetchone()

cursor.execute("select color, quest from knights")
rows = cursor.fetchall()

cursor.execute("select * from huge_table)
while True:
    rows = cursor.fetchmany(1000)
    if not rows:
        break
    for row in rows:
        # process row
```


Example

db_sqlite_basics.py

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

import sqlite3

with sqlite3.connect("../DATA/presidents.db") as s3conn:  ①

    s3cursor = s3conn.cursor()  ②

    # select first name, last name from all presidents
    s3cursor.execute('''
        select firstname, lastname
        from presidents
    ''')  ③

    print("Sqlite3 does not provide a row count\n")  ④

    for row in s3cursor.fetchall():  ⑤
        print(' '.join(row))  ⑥
```

① connect to the database

② get a cursor object

③ execute a SQL statement

④ (included for consistency with other DBMS modules)

⑤ fetchall() returns all rows

⑥ each row is a tuple

db_sqlite_basics.py

```
Richard Milhous Nixon
Gerald Rudolph Ford
James Earl 'Jimmy' Carter
Ronald Wilson Reagan
George Herbert Walker Bush
William Jefferson 'Bill' Clinton
George Walker Bush
Barack Hussein Obama
Donald J Trump
Joseph Robinette Biden
```


NOTE See db_mysql_basics.py and db_postgres_basics.py for examples using those modules. In general, all the sqlite3 examples are also implemented for MySQL and Postgres, plus a few extras.

SQL Injection

- "Hijacks" SQL code
- Result of string formatting
- Always use parameterized statements

One kind of vulnerability in SQL code is called SQL injection. This occurs when an attacker embeds SQL commands in input data. This can happen when naively using string formatting to build SQL statements:

Example

db_sql_injection.py

```
#!/usr/bin/env python
#
good_input = 'Google'
malicious_input = "'; drop table customers; -- "  ①

naive_format = "select * from customers where company_name = '{}' and company_id != 0"

good_query = naive_format.format(good_input)  ②
malicious_query = naive_format.format(malicious_input)  ②

print("Good query:")
print(good_query)  ③
print()

print("Bad query:")
print(malicious_query)  ④
```

① input would come from a web form, for instance

② string formatting naively adds the user input to a field, expecting only a customer name

③ non-malicious input works fine

④ query now drops a table ('--' is SQL comment) *db_sql_injection.py*

```
Good query:
select * from customers where company_name = 'Google' and company_id != 0

Bad query:
select * from customers where company_name = ''; drop table customers; -- ' and
company_id != 0
```

Parameterized Statements

- More efficient updates
- Use placeholders in query
  - Placeholders vary by DB
- Pass iterable of parameters
- Prevent SQL injection
- Use cursor.execute() or cursor.executemany()

For efficiency, you can iterate over of sequence of input datasets when performing a non-query SQL statement. The execute() method takes a query, plus an iterable of values to fill in the placeholders. The database manager will only parse the query once, then reuse it for subsequent calls to execute().

Parameterized queries also protect against SQL injection attacks.

Different database modules use different placeholders. To see what kind of placeholder a module uses, check MODULE.paramstyle. Types include 'pyformat', meaning '%s', and 'qmark', meaning '?'.

The executemany() method takes a query, plus an iterable of iterables. It will call execute() once for each nested iterable.

Example

```
single_row = ("Smith","John","green"),

multi_rows= [
    ("Smith","John","green"),
    ("Douglas","Sam","pink"),
    ("Robinson","Alberta","blue"),
]

query = "insert into people (lname,fname,color) values (%s,%s,%s)"

rows_added = cursor.execute(query, single_row)
rows_added = cursor.executemany(query, multi_rows)
```

Table 8. Placeholders for SQL Parameters

Python package	Placeholder for parameters
pymysql	%s
cx_oracle	:param_name
pyodbc	?
pymssql	%d for int, %s for str, etc.
Psychopg	%s or %(param_name)s
sqlite3	? or :param_name

TIP with the exception of **pymssql** the same placeholder is used for all column types.

Example

db_sqlite_parameterized.py

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

import sqlite3

with sqlite3.connect("../DATA/presidents.db") as s3conn:
    s3cursor = s3conn.cursor()

    party_query = '''
    select firstname, lastname
    from presidents
        where party = ?
    '''  ①

    for party in 'Federalist', 'Whig':
        print(party)
        s3cursor.execute(party_query, (party,))  ②
        print(s3cursor.fetchall())
        print()
```

① ? is SQLite3 placeholder for SQL statement parameter; different DBMSs use different placeholders

② second argument to execute() is iterable of values to fill in placeholders from left to right

db_sqlite_parameterized.py

```
Federalist
[('John', 'Adams')]

Whig
[('William Henry', 'Harrison'), ('John', 'Tyler'), ('Zachary', 'Taylor'), ('Millard',
'Fillmore')]
```

Example

db_sqlite_bulk_insert.py

```
#!/usr/bin/env python
import os
import sqlite3
import random

FRUITS = ["pomegranate", "cherry", "apricot", "date", "apple",
          "lemon", "kiwi", "orange", "lime", "watermelon", "guava",
          "papaya", "fig", "pear", "banana", "tamarind", "persimmon",
          "elderberry", "peach", "blueberry", "lychee", "grape"]

DB_NAME = 'fruitprices.db'  ①

CREATE_TABLE = """
create table fruit (
    name varchar(30),
    price decimal
)
"""  ②

INSERT = '''
insert into fruit (name, price) values (?, ?)
'''  ③


def main():
    """
    Program entry point.

    :return: None
    """
    conn = get_connection()
    create_database(conn)
    populate_database(conn)

    read_database()


def get_connection():
    """
    Get a connection to the PRODUCE database

    :return: SQLite3 connection object.
    """
    if os.path.exists(DB_NAME):
```



```
        os.remove(DB_NAME)  ④

    s3conn = sqlite3.connect(DB_NAME)  ⑤
    return s3conn


def create_database(conn):
    """
    Create the fruit table

    :param conn: The database connection
    :return: None
    """
    conn.execute(CREATE_TABLE)  ⑥


def populate_database(conn):
    """
    Add rows to the fruit table

    :param conn: The database connection
    :return: None
    """

    fruit_data = get_fruit_data()  # [('apple', .49), ('kiwi', .38)]

    try:
        conn.executemany(INSERT, fruit_data)  ⑦
    except sqlite3.DatabaseError as err:
        print(err)
        conn.rollback()
    else:
        conn.commit()  ⑧


def get_fruit_data():
    """
    Create iterable of fruit records.

    :return: Generator of name/price tuples.
    """
    return ((f, round(random.random() * 10 + 5, 2)) for f in FRUITS)  ⑨


def read_database():
    conn = sqlite3.connect(DB_NAME)
    for name, price in conn.execute('select name, price from fruit'):
        print('{:12s} {:6.2f}'.format(name, price))
```

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

① set name of database

② SQL statement to create table

③ parameterized SQL statement to insert one record

④ remove existing database if it exists

⑤ connect to (new) database

⑥ run SQL to create table

⑦ iterate over list of pairs and add each pair to the database

⑧ commit the inserts; without this, no data would be saved

⑨ build list of tuples containing fruit, price pairs

db_sqlite_bulk_insert.py

```
pomegranate   12.40
cherry         7.92
apricot        6.47
date          10.94
apple         14.06
lemon         10.03
kiwi           8.78
orange        13.91
lime          13.31
watermelon     8.88
guava         14.17
papaya        10.62
fig            9.59
pear          12.64
banana        14.45
tamarind      14.86
persimmon     11.56
elderberry     8.04
peach         12.85
blueberry     11.92
lychee        10.95
grape          6.48
```


Dictionary Cursors

- Indexed by column name
- Not standardized in the DB API

The standard cursor provided by the DB API returns a tuple for each row. Most DB packages provide other kinds of cursors, including user-defined versions.

A very common cursor is a dictionary cursor, which returns a dictionary for each row, where the keys are the column names. Each package that provides a dictionary cursor has its own way of providing the dictionary cursor, although they all work the same way.

For the packages that don't have a dictionary cursor, you can make a generator function that will emulate one.

Table 9. Dictionary Cursors

Python package	How to get a dictionary cursor
pymysql	`import pymysql.cursors` `conn = pymysql.connect(…,` `cursorclass = pymysql.cursors.DictCursor` `)` `dcur = conn.cursor()` *all cursors will be dict cursors* `dcur = conn.cursor( pymysql.cursors.DictCursor)` *only this cursor will be a dict cursor*
cx_oracle	*Not available*
pyodbc	*Not available*
pgdb	*Not available*
pymssql	`conn = pymssql.connect (…, as_dict=True)` `dcur = conn.cursor()`
psychopg	`import psycopg2.extras` `dcur =` `conn.cursor(cursor_factory=psycopg.extras.DictCursor)`
sqlite3	`conn = sqlite3.connect (…,` `row_factory=sqlite3.Row)` `dcur = conn.cursor()` `conn.row_factory = sqlite3.Row` `dcur = conn.cursor()`

Example

db_sqlite_extras.py

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

import sqlite3

s3conn = sqlite3.connect("../DATA/presidents.db")
# uncomment to make _all_ cursors dictionary cursors
# conn.row_factory = sqlite3.Row

NAME_QUERY = '''
    select firstname, lastname
    from presidents
    where termnum < 5
'''

cur = s3conn.cursor()

# select first name, last name from all presidents
cur.execute(NAME_QUERY)

for row in cur.fetchall():
    print(row)
print('-' * 50)

dcur = s3conn.cursor()  ①

# make _this_ cursor a dictionary cursor
dcur.row_factory = sqlite3.Row  ②

# select first name, last name from all presidents
dcur.execute(NAME_QUERY)

for row in dcur.fetchall():
    print(row['firstname'], row['lastname'])  ③

print('-' * 50)
```

① default cursor returns tuple for each row

② Row object is tuple/dict hybrid; can be indexed by position OR column name

③ selecting by column name

db_sqlite_extras.py

```
('George', 'Washington')
('John', 'Adams')
('Thomas', 'Jefferson')
('James', 'Madison')
--------------------------------------------------
George Washington
John Adams
Thomas Jefferson
James Madison
--------------------------------------------------
```


Metadata

- cursor.description returns tuple of tuples
- Fields
    - name
    - type_code
    - display_size
    - internal_size
    - precision
    - scale
    - null_ok

Once a query has been executed, the cursor's description attribute is a tuple with metadata about the columns in the query. It contains one tuple for each column in the query, containing 7 values describing the column.

For instance, to get the names of the columns, you could say:

```
names = [ d[0] for d in cursor.description ]
```

For non-query statements, cursor.description returns None.

The names are based on the query (with possible aliases), and not necessarily on the names in the table.

Example

NOTE Many database modules, including pymysql, have a dictionary cursor built in — this is just for an example you could use with any DB API module that does not have this capability. The example uses the metadata from the cursor to get the column names, and forms a dictionary by zipping the column names with the column values. Another approach would be to use a named tuple. __

db_sqlite_emulate_dict_cursor.py

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

import sqlite3

s3conn = sqlite3.connect("../DATA/presidents.db")

c = s3conn.cursor()


def row_as_dict(cursor):
    '''Generate rows as dictionaries'''
    column_names = [desc[0] for desc in cursor.description]
    for cursor_row in cursor.fetchall():
        row_dict = dict(zip(column_names, cursor_row))
        yield row_dict


# select first name, last name from all presidents
num_recs = c.execute('''
    select lastname, firstname
    from presidents
''')

for row in row_as_dict(c):
    print(row['firstname'], row['lastname'])
```


db_sqlite_emulate_dict_cursor.py

```
Richard Milhous Nixon
Gerald Rudolph Ford
James Earl 'Jimmy' Carter
Ronald Wilson Reagan
George Herbert Walker Bush
William Jefferson 'Bill' Clinton
George Walker Bush
Barack Hussein Obama
Donald J Trump
Joseph Robinette Biden
```

See db_sqlite_named_tuple_cursor.py for a similar example that creates named tuples rather than dictionaries for each row.

Transactions

- Transactions allow safer control of updates
- commit() to save transactions
- rollback() to discard
- Default is autocommit off
- autocommit\=True to turn on

Sometimes a database task involves more than one change to your database (i.e., more than one SQL statement). You don't want the first SQL statement to succeed and the second to fail; this would leave your database in a corrupt state.

To be certain of data integrity, use **transactions**. This lets you make multiple changes to your database and only commit the changes if all the SQL statements were successful. For all packages using the Python DB API, a transaction is started when you connect. At any point, you can call *CONNECTION*.commit() to save the changes, or *CONNECTION*.rollback() to discard the changes. For most packages, if you don't call commit() after modify a table, the data will not be saved.

NOTE You can also turn on autocommit, which calls commit() after every statement.

Example

```
try:
    for info in list_of_tuples:
         cursor.execute(query,info)
except SQLError:
    dbconn.rollback()
else:
    dbconn.commit()
```

NOTE **pymysql** only supports transaction processing when using the **InnoDB** engine

Object-relational Mappers

- No SQL required
- Maps a class to a table
- All DB work is done by manipulating objects
- Most popular Python ORMs
    - SQLAlchemy
    - Django (which is a complete web framework)

An Object-relational mapper is a module or framework that creates a level of abstraction above the actual database tables and SQL queries. As the name implies, a Python class (object) is mapped to the actual table.

The two most popular Python ORMs are SQLAlchemy which is a standalone ORM, and Django ORM. Django is a comprehensive Web development framework, which provides an ORM as a subpackage. SQLAlchemy is the most fully developed package, and is the ORM used by Flask and some other Web development frameworks.

Instead of querying the database, you call a search method on an object representing a table. To add a row to the table, you create a new instance of the table class, populate it, and call a method like save(). You can create a large, complex database system, complete with foreign keys, composite indices, and all the other attributes near and dear to a DBA, without writing the first line of SQL.

You can use Python ORMs in two ways.

One way is to design the database with the ORM. To do this, you create a class for each table in the database, specifying the columns with predefined classes from the ORM. Then you run an ORM command which executes the queries needed to build the database. If you need to make changes, you update the class definitions, and run an ORM command to synchronize the actual DBMS to your classes.

The second way is to map tables to an existing database. You create the classes to match the schemas that have already been defined in the database. Both SQLAlchemy and the Django ORM have tools to automate this process.

NoSQL

- Non-relational database
- Document-oriented
- Can be hierarchical (nested)
- Examples
    - MongoDB
    - Cassandra
    - Redis

A current trend in data storage are called "NoSQL" or non-relational databases. These databases consist of *documents*, which are indexed, and may contain nested data.

NoSQL databases don't contain tables, and do not have relations.

While relational databases are great for tabular data, they are not as good a fit for nested data. Geospatial, engineering diagrams, and molecular modeling can have very complex structures. It is possible to shoehorn such data into a relational database, but a NoSQL database might work much better. Another advantage of NoSQL is that it can adapt to changing data structures, without having to rebuild tables if columns are added, deleted, or modified.

Some of the most common NoSQL database systems are MongoDB, Cassandra and Redis.

Example

mongodb_example.py

```
#!/usr/bin/env python
import re
from pymongo import MongoClient, errors

FIELD_NAMES = (
    'termnumber lastname firstname '
    'birthdate '
    'deathdate birthplace birthstate '
    'termstartdate '
    'termenddate '
    'party'
).split()  ①

mc = MongoClient()  ②

try:
    mc.drop_database("presidents")  ③
except errors.PyMongoError as err:
    print(err)

db = mc["presidents"]  ④

coll = db.presidents  ⑤

with open('../DATA/presidents.txt') as presidents_in:  ⑥
    for line in presidents_in:
        flds = line[:-1].split(':')
        kvpairs = zip(FIELD_NAMES, flds)
        record_dict = dict(kvpairs)
        coll.insert_one(record_dict)  ⑦

print(db.list_collection_names())  ⑧
print()

abe = coll.find_one({'termnumber': '16'})  ⑨
print(abe, '\n')

for field in FIELD_NAMES:
    print("{0:15s} {1}".format(field.upper(), abe[field]))  ⑩

print('-' * 50)

for president in coll.find():  ⑪
    print("{0[firstname]:25s} {0[lastname]:30s}".format(president))
```

```
print('-' * 50)

rx_lastname = re.compile('^roo', re.IGNORECASE)
for president in coll.find({'lastname': rx_lastname}):  ⑫
    print("{0[firstname]:25s} {0[lastname]:30s}".format(president))
print('-' * 50)

for president in coll.find({"birthstate": 'Virginia'}):  ⑬
    print("{0[firstname]:25s} {0[lastname]:30s}".format(president))

print('-' * 50)
print("removing Millard Fillmore")
result = coll.delete_one({'lastname': 'Fillmore'})  ⑭
print(result)
result = coll.delete_one({'lastname': 'Roosevelt'})  ⑭
print(result)
print('-' * 50)


result = coll.delete_one({'lastname': 'Bush'})
print(dir(result))
print()

result = coll.count_documents({})  ⑮
print(result)


for president in coll.find():  ⑪
    print("{0[firstname]:25s} {0[lastname]:30s}".format(president))
print('-' * 50)

animals = db.animals

print(animals, '\n')

animals.insert_one({'name': 'wombat', 'country': 'Australia'})
animals.insert_one({'name': 'ocelot', 'country': 'Mexico'})
animals.insert_one({'name': 'honey badger', 'country': 'Iran'})

for doc in animals.find():
    print(doc['name'])
```

① define some field name

② get a Mongo client

③ delete 'presidents' database if it exists

④ create a new database named 'presidents'

⑤ get the collection from presidents db

⑥ open a data file

⑦ insert a record into collection

⑧ get list of collections

⑨ search collection for doc where termnumber == 16

⑩ print all fields for one record

⑪ loop through all records in collection

⑫ find record using regular expression

⑬ find record searching multiple fields

⑭ delete record

⑮ get count of records

mongodb_example.py

```
William Howard           Taft
Woodrow                  Wilson
Warren Gamaliel          Harding
Calvin                   Coolidge
Herbert Clark            Hoover
Franklin Delano          Roosevelt
Harry S.                 Truman
Dwight David             Eisenhower
John Fitzgerald          Kennedy
Lyndon Baines            Johnson
Richard Milhous          Nixon
Gerald Rudolph           Ford
James Earl 'Jimmy'       Carter
Ronald Wilson            Reagan
William Jefferson 'Bill' Clinton
George Walker            Bush
Barack Hussein           Obama
Donald John              Trump
Joseph Robinette         Biden
--------------------------------------------------
Collection(Database(MongoClient(host=['localhost:27017'], document_class=dict,
tz_aware=False, connect=True), 'presidents'), 'animals')

wombat
ocelot
honey badger
```

Chapter 4 Exercises

Exercise 4-1 (president_sqlite.py)

For this exercise, you can use the SQLite3 database provided, or use your own DBMS. The mkpres.sql script is generic and should work with any DBMS to create and populate the presidents table. The SQLite3 database is named **presidents.db** and is located in the DATA folder of the student files.

The data has the following layout

Table 10. Layout of President Table

Field Name	Data Type	Null	Default
termnum	int(11)	YES	NULL
lastname	varchar(32)	YES	NULL
firstname	varchar(64)	YES	NULL
termstart	date	YES	NULL
termend	date	YES	NULL
birthplace	varchar(128)	YES	NULL
birthstate	varchar(32)	YES	NULL
birthdate	date	YES	NULL
deathdate	date	YES	NULL
party	varchar(32)	YES	NULL

Refactor the **president.py** module to get its data from this table, rather than from a file. Re-run your previous scripts that used president.py; now they should get their data from the database, rather than from the flat file.

NOTE If you created a president.py module as part of an earlier lab, use that. Otherwise, use the supplied president.py module in the top folder of the student files.

Exercise 4-2 (add_pres_sqlite.py)

Add the next president to the presidents database. Just make up the data —let's keep this non-political. Don't use any real-life people.

SQL syntax for adding a record is

```
INSERT INTO table ("COL1-NAME",..) VALUES ("VALUE1",...)
```

To do a parameterized insert (the right way!):

```
INSERT INTO table ("COL1-NAME",..) VALUES (%s,%s,...)  # MySQL
INSERT INTO table ("COL1-NAME",..) VALUES (?,?,...)    # SQLite
```

or whatever your database uses as placeholders

NOTE There are also MySQL versions of the answers.

Chapter 5: Effective Scripts

Objectives

- Launch external programs
- Check permissions on files
- Get system configuration information
- Store data offline
- Create Unix-style filters
- Parse command line options
- Configure application logging

Using glob

- Expands wildcards
- Windows and non-windows
- Useful with **subprocess** module

When executing external programs, sometimes you want to specify a list of files using a wildcard. The **glob** function in the **glob** module will do this. Pass one string containing a wildcard (such as `*.txt`) to glob(), and it returns a sorted list of the matching files. If no files match, it returns an empty list.

Example

glob_example.py

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

from glob import glob

files = glob('../DATA/*.txt') ①
print(files, '\n')

no_files = glob('../JUNK/*.avi')
print(no_files, '\n')
```

① expand file name wildcard into sorted list of matching names *glob_example.py*

```
['../DATA/presidents_plus_biden.txt', '../DATA/columns_of_numbers.txt',
'../DATA/poe_sonnet.txt', '../DATA/computer_people.txt', '../DATA/owl.txt',
'../DATA/eggs.txt', '../DATA/world_airport_codes.txt', '../DATA/stateinfo.txt',
'../DATA/fruit2.txt', '../DATA/us_airport_codes.txt', '../DATA/parrot.txt',
'../DATA/http_status_codes.txt', '../DATA/fruit1.txt', '../DATA/alice.txt',
'../DATA/littlewomen.txt', '../DATA/spam.txt', '../DATA/world_median_ages.txt',
'../DATA/phone_numbers.txt', '../DATA/sales_by_month.txt', '../DATA/engineers.txt',
'../DATA/underrated.txt', '../DATA/tolkien.txt', '../DATA/tyger.txt',
'../DATA/example_data.txt', '../DATA/states.txt', '../DATA/kjv.txt', '../DATA/fruit.txt',
'../DATA/areacodes.txt', '../DATA/float_values.txt', '../DATA/unabom.txt',
'../DATA/chaos.txt', '../DATA/noisewords.txt', '../DATA/presidents.txt',
'../DATA/bible.txt', '../DATA/breakfast.txt', '../DATA/Pride_and_Prejudice.txt',
'../DATA/nsfw_words.txt', '../DATA/mary.txt',
'../DATA/2017FullMembersMontanaLegislators.txt', '../DATA/badger.txt',
'../DATA/README.txt', '../DATA/words.txt', '../DATA/primeministers.txt',
'../DATA/grail.txt', '../DATA/alt.txt', '../DATA/knights.txt',
'../DATA/world_airports_codes_raw.txt', '../DATA/correspondence.txt']

[]
```

Using shlex.split()

- Splits string
- Preserves white space

If you have an external command you want to execute, you should split it into individual words. If your command has quoted whitespace, the normal **split()** method of a string won't work.

For this you can use **shlex.split()**, which preserves quoted whitespace within a string.

Example

shlex_split.py

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

cmd = 'herp derp "fuzzy bear" "wanga tanga" pop'  ①

print(cmd.split())  ②
print()

print(shlex.split(cmd))  ③
```

① Command line with quoted whitespace

② Normal split does the wrong thing

③ shlex.split() does the right thing

shlex_split.py

```
['herp', 'derp', '"fuzzy', 'bear"', '"wanga', 'tanga"', 'pop']

['herp', 'derp', 'fuzzy bear', 'wanga tanga', 'pop']
```


The subprocess module

- Spawns new processes
- works on Windows and non-Windows systems
- Convenience methods
  - **run()**
  - **call()**, **check_call()**

The **subprocess** module spawns and manages new processes. You can use this to run local non-Python programs, to log into remote systems, and generally to execute command lines.

subprocess implements a low-level class named Popen; However, the convenience methods **run()**, **check_call()**, and check_output()**, which are built on top of Popen(), are commonly used, as they have a simpler interface. You can capture *stdout** and **stderr**, separately. If you don't capture them, they will go to the console.

In all cases, you pass in an iterable containing the command split into individual words, including any file names. This is why this chapter starts with glob.glob() and shlex.split().

Table 11. CalledProcessError attributes

Attribute	Description
args	The arguments used to launch the process. This may be a list or a string.
returncode	Exit status of the child process. Typically, an exit status of 0 indicates that it ran successfully. A negative value -N indicates that the child was terminated by signal N (POSIX only).
stdout	Captured stdout from the child process. A bytes sequence, or a string if run() was called with an encoding or errors. None if stdout was not captured. If you ran the process with stderr=subprocess.STDOUT, stdout and stderr will be combined in this attribute, and stderr will be None. stderr

subprocess convenience functions

- run(), check_call() , check_output()
- Simpler to use than Popen

subprocess defines convenience functions, **call()**, **check_call()**, and **check_output()**.

```
proc subprocess.run(cmd, ...)
```

Run command with arguments. Wait for command to complete, then return a **CompletedProcess** instance.

```
subprocess.check_call(cmd, ...)
```

Run command with arguments. Wait for command to complete. If the exit code was zero then return, otherwise raise CalledProcessError. The CalledProcessError object will have the return code in the returncode attribute.

```
check_output(cmd, ...)
```

Run command with arguments and return its output as a byte string. If the exit code was non-zero it raises a CalledProcessError. The CalledProcessError object will have the return code in the returncode attribute and output in the output attribute.

NOTE run() is only implemented in Python 3.5 and later.

Example

subprocess_conv.py

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

import sys
from subprocess import check_call, check_output, CalledProcessError
from glob import glob
import shlex

if sys.platform == 'win32':
    CMD = 'cmd /c dir'
    FILES = r'..\DATA\t*'
else:
    CMD = 'ls -ld'
    FILES = '../DATA/t*'

cmd_words = shlex.split(CMD)
cmd_files = glob(FILES)

full_cmd = cmd_words + cmd_files

try:
    check_call(full_cmd)
except CalledProcessError as err:
    print("Command failed with return code", err.returncode)

print('-' * 60)

try:
    output = check_output(full_cmd)
    print("Output:", output.decode(), sep='\n')
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)
```

subprocess_conv.py

```
-rw-r--r--  1 jstrick  staff  3178541 Nov  2 09:27 ../DATA/tate_data.zip
-rwxr-xr-x  1 jstrick  staff      297 Nov 17  2016 ../DATA/testscores.dat
-rwxr-xr-x  1 jstrick  staff     2198 Feb 14  2016 ../DATA/textfiles.zip
-rw-r--r--  1 jstrick  staff   106960 Jul 26  2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick  staff   284160 Jul 26  2017 ../DATA/titanic3.xls
-rwxr-xr-x  1 jstrick  staff    73808 Feb 14  2016 ../DATA/tolkien.txt
-rwxr-xr-x  1 jstrick  staff      834 Feb 14  2016 ../DATA/tyger.txt
------------------------------------------------------------
Output:
-rw-r--r--  1 jstrick  staff  3178541 Nov  2 09:27 ../DATA/tate_data.zip
-rwxr-xr-x  1 jstrick  staff      297 Nov 17  2016 ../DATA/testscores.dat
-rwxr-xr-x  1 jstrick  staff     2198 Feb 14  2016 ../DATA/textfiles.zip
-rw-r--r--  1 jstrick  staff   106960 Jul 26  2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick  staff   284160 Jul 26  2017 ../DATA/titanic3.xls
-rwxr-xr-x  1 jstrick  staff    73808 Feb 14  2016 ../DATA/tolkien.txt
-rwxr-xr-x  1 jstrick  staff      834 Feb 14  2016 ../DATA/tyger.txt

--------------------------------------------------
```

NOTE showing Unix/Linux/Mac output – Windows will be similar

TIP (Windows only) The following commands are *internal* to CMD.EXE, and must be preceded by `cmd /c` or they will not work: ASSOC, BREAK, CALL ,CD/CHDIR, CLS, COLOR, COPY, DATE, DEL, DIR, DPATH, ECHO, ENDLOCAL, ERASE, EXIT, FOR, FTYPE, GOTO, IF, KEYS, MD/MKDIR, MKLINK (vista and above), MOVE, PATH, PAUSE, POPD, PROMPT, PUSHD, REM, REN/RENAME, RD/RMDIR, SET, SETLOCAL, SHIFT, START, TIME, TITLE, TYPE, VER, VERIFY, VOL

Capturing stdout and stderr

- Add stdout, stderr args
- Assign subprocess.PIPE

To capture stdout and stderr with the subprocess module, import **PIPE** from subprocess and assign it to the stdout and stderr parameters to run(), check_call(), or check_output(), as needed.

For check_output(), the return value is the standard output; for run(), you can access the **stdout** and **stderr** attributes of the CompletedProcess instance returned by run().

NOTE output is returned as a bytes object; call decode() to turn it into a normal Python string.

Example

subprocess_capture.py

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

import sys
from subprocess import check_output, Popen, CalledProcessError, STDOUT, PIPE ①
from glob import glob
import shlex

if sys.platform == 'win32':
    CMD = 'cmd /c dir'
    FILES = r'..\DATA\t*'
else:
    CMD = 'ls -ld'
    FILES = '../DATA/t*'

cmd_words = shlex.split(CMD)
cmd_files = glob(FILES)

full_cmd = cmd_words + cmd_files

②
try:
    output = check_output(full_cmd) ③
    print("Output:", output.decode(), sep='\n') ④
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)
```

```
⑤
try:
    cmd = cmd_words + cmd_files + ['spam.txt']
    proc = Popen(cmd, stdout=PIPE, stderr=STDOUT) ⑥
    stdout, stderr = proc.communicate() ⑦
    print("Output:", stdout.decode()) ⑧
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)

try:
    cmd = cmd_words + cmd_files + ['spam.txt']
    proc = Popen(cmd, stdout=PIPE, stderr=PIPE) ⑨
    stdout, stderr = proc.communicate() ⑩
    print("Output:", stdout.decode()) ⑪
    print("Error:", stderr.decode()) ⑪
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)
```

① need to import PIPE and STDOUT

② capture only stdout

③ check_output() returns stdout

④ stdout is returned as bytes (decode to str)

⑤ capture stdout and stderr together

⑥ assign PIPE to stdout, so it is captured; assign STDOUT to stderr, so both are captured together

⑦ call communicate to get the input streams of the process; it returns two bytes objects representing stdout and stderr

⑧ decode the stdout object to a string

⑨ assign PIPE to stdout and PIPE to stderr, so both are captured individually

⑩ now stdout and stderr each have data

⑪ decode from bytes and output

subprocess_capture.py

```
Output:
-rw-r--r--  1 jstrick  staff  3178541 Nov  2 09:27 ../DATA/tate_data.zip
-rwxr-xr-x  1 jstrick  staff      297 Nov 17  2016 ../DATA/testscores.dat
-rwxr-xr-x  1 jstrick  staff     2198 Feb 14  2016 ../DATA/textfiles.zip
-rw-r--r--  1 jstrick  staff   106960 Jul 26  2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick  staff   284160 Jul 26  2017 ../DATA/titanic3.xls
-rwxr-xr-x  1 jstrick  staff    73808 Feb 14  2016 ../DATA/tolkien.txt
-rwxr-xr-x  1 jstrick  staff      834 Feb 14  2016 ../DATA/tyger.txt

--------------------------------------------------
Output: -rw-r--r--  1 jstrick  staff     3178541 Nov  2 09:27 ../DATA/tate_data.zip
-rwxr-xr-x  1 jstrick  staff         297 Nov 17  2016 ../DATA/testscores.dat
-rwxr-xr-x  1 jstrick  staff        2198 Feb 14  2016 ../DATA/textfiles.zip
-rw-r--r--  1 jstrick  staff      106960 Jul 26  2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick  staff      284160 Jul 26  2017 ../DATA/titanic3.xls
-rwxr-xr-x  1 jstrick  staff       73808 Feb 14  2016 ../DATA/tolkien.txt
-rwxr-xr-x  1 jstrick  staff         834 Feb 14  2016 ../DATA/tyger.txt
-rw-r--r--  1 jstrick  students       22 Nov 24 09:05 spam.txt

--------------------------------------------------
Output: -rw-r--r--  1 jstrick  staff     3178541 Nov  2 09:27 ../DATA/tate_data.zip
-rwxr-xr-x  1 jstrick  staff         297 Nov 17  2016 ../DATA/testscores.dat
-rwxr-xr-x  1 jstrick  staff        2198 Feb 14  2016 ../DATA/textfiles.zip
-rw-r--r--  1 jstrick  staff      106960 Jul 26  2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick  staff      284160 Jul 26  2017 ../DATA/titanic3.xls
-rwxr-xr-x  1 jstrick  staff       73808 Feb 14  2016 ../DATA/tolkien.txt
-rwxr-xr-x  1 jstrick  staff         834 Feb 14  2016 ../DATA/tyger.txt
-rw-r--r--  1 jstrick  students       22 Nov 24 09:05 spam.txt

Error:
--------------------------------------------------
```

Permissions

- Simplest is os.access()
- Get mode from os.lstat()
- Use binary AND with permission constants

Each entry in a Unix filesystem has a inode. The inode contains low-level information for the file, directory, or other filesystem entity. Permissions are stored in the 'mode', which is a 16-bit unsigned integer. The first 4 bits indicate what kind of entry it is, and the last 12 bits are the permissions.

To see if a file or directory is readable, writable, or executable use os.access(). To test for specific permissions, use the os.lstat() method to return a tuple of inode data, and use the S_IMODE () method to get the mode information as a number. Then use predefined constants such as stat.S_IRUSR, stat.S_IWGRP, etc. to test for permissions.

Example

file_access.py

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

import sys
import os

if len(sys.argv) < 2:
    start_dir = "."
else:
    start_dir = sys.argv[1]

for base_name in os.listdir(start_dir):  ①
    file_name = os.path.join(start_dir, base_name)
    if os.access(file_name, os.W_OK):  ②
        print(file_name, "is writable")
```

① os.listdir() lists the contents of a directory

② os.access() returns True if file has specified permissions (can be os.W_OK, os.R_OK, or os.X_OK, combined with | (OR))

file_access.py ../DATA

```
../DATA/presidents.csv is writable
../DATA/wetprf is writable
../DATA/uri-schemes-1.csv is writable
../DATA/presidents.html is writable
../DATA/presidents.xlsx is writable
../DATA/presidents_plus_biden.txt is writable
../DATA/baby_names is writable
../DATA/presidents.db is writable
../DATA/testscores.dat is writable
../DATA/solar.json is writable
```

…

Using shutil

- Portable ways to copy, move, and delete files
- Create archives
- Misc utilities

The **shutil** module provides portable functions for copying, moving, renaming, and deleting files. There are several variations of each command, depending on whether you need to copy all the attributes of a file, for instance.

The module also provides an easy way to create a zip file or compressed **tar** archive of a folder.

In addition, there are some miscellaneous convenience routines.

Example

shutil_ex.py

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

shutil.copy('../DATA/alice.txt', 'betsy.txt') ①

print("betsy.txt exists:", os.path.exists('betsy.txt'))

shutil.move('betsy.txt', 'fred.txt') ②
print("betsy.txt exists:", os.path.exists('betsy.txt'))
print("fred.txt exists:", os.path.exists('fred.txt'))

new_folder = 'remove_me'

os.mkdir(new_folder) ③
shutil.move('fred.txt', new_folder)

shutil.make_archive(new_folder, 'zip', new_folder) ④

print("{}.zip exists:".format(new_folder), os.path.exists(new_folder + '.zip'))

print("{} exists:".format(new_folder), os.path.exists(new_folder))

shutil.rmtree(new_folder) ⑤

print("{} exists:".format(new_folder), os.path.exists(new_folder))
```

① copy file

② rename file

③ create new folder

④ make a zip archive of new folder

⑤ recursively remove folder

shutil_ex.py

```
betsy.txt exists: True
betsy.txt exists: False
fred.txt exists: True
remove_me.zip exists: True
remove_me exists: True
remove_me exists: False
```


Creating a useful command line script

- More than just some lines of code
- Input + Business Logic + Output
- Process files for input, or STDIN
- Allow options for customizing execution
- Log results

A good system administration script is more than just some lines of code hacked together. It needs to gather data, apply the appropriate business logic, and, if necessary, output the results of the business logic to the desired destination.

Python has two tools in the standard library to help create professional command line scripts. One of these is the argparse module, for parsing options and parameters on the script's command line. The other is fileinput, which simplifies processing a list of files specified on the command line.

We will also look at the logging module, which can be used in any application to output to a variety of log destinations, including a plain file, syslog on Unix-like systems or the NTLog service on Windows, or even email.

Creating filters

- Filter reads files or STDIN and writes to STDOUT

> Common on Unix systems Well-known filters: awk, sed, grep, head, tail, cat Reads command line arguments as files, otherwise STDIN use fileinput.input()

A common kind of script iterates over all lines in all files specified on the command line. The algorithm is

```
for filename in sys.argv[1:]:
    with open(filename) as F:
       for line in F:
            # process line
```

Many Unix utilities are written to work this way – sed, grep, awk, head, tail, sort, and many more. They are called filters, because they filter their input in some way and output the modified text. Such filters read STDIN if no files are specified, so that they can be piped into.

The fileinput.input() class provides a shortcut for this kind of file processing. It implicitly loops through sys.argv[1:], opening and closing each file as needed, and then loops through the lines of each file. If sys.argv[1:] is empty, it reads sys.stdin. If a filename in the list is '-', it also reads sys.stdin.

fileinput works on Windows as well as Unix and Unix-like platforms.

To loop through a different list of files, pass an iterable object as the argument to fileinput.input().

There are several methods that you can call from fileinput to get the name of the current file, e.g.

Table 12. fileinput Methods

Method	Description
filename()	Name of current file being readable
lineno()	Cumulative line number from all files read so far
filelineno()	Line number of current file
isfirstline()	True if current line is first line of a file
isstdin()	True if current file is sys.stdin
close()	Close fileinput

Example

file_input.py

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

import fileinput

for line in fileinput.input():  ①
    if 'bird' in line:
        print('{}: {}'.format(fileinput.filename(), line), end=' ')  ②
```

① fileinput.input() is a generator of all lines in all files in sys.argv[1:]

② fileinput.filename() has the name of the current file

file_input.py ../DATA/parrot.txt ../DATA/alice.txt

```
../DATA/parrot.txt: At that point, the guy is so mad that he throws the bird into the
../DATA/parrot.txt: For the first few seconds there is a terrible din. The bird kicks
../DATA/parrot.txt: bird may be hurt. After a couple of minutes of silence, he's so
../DATA/parrot.txt: The bird calmly climbs onto the man's out-stretched arm and says,
../DATA/alice.txt: with the birds and animals that had fallen into it:  there were a
../DATA/alice.txt: bank--the birds with draggled feathers, the animals with their
../DATA/alice.txt: some of the other birds tittered audibly.
../DATA/alice.txt: and confusion, as the large birds complained that they could not
```


Parsing the command line

- Parse and analyze sys.argv
- use argparse
- parses entire command line
- very flexible
- validates options and arguments

Many command line scripts need to accept options and arguments. In general, options control the behavior of the script, while arguments provide input. Arguments are frequently file names, but can be anything. All arguments are available in Python via sys.argv

There are at least three modules in the standard library to parse command line options. The oldest module is getopt (earlier than v1.3), then optparse (introduced 2.3, now deprecated), and now, argparse is the latest and greatest. (Note: argparse is only available in 2.7 and 3.0+).

To get started with argparse, create an ArgumentParser object. Then, for each option or argument, call the parser’s add_argument() method.

The add_argument() method accepts the name of the option (e.g. '-count') or the argument (e.g. 'filename'), plus named parameters to configure the option.

Once all arguments have been described, call the parser’s parse_args() method. (By default, it will process sys.argv, but you can pass in any list or tuple instead.) parse_args() returns an object containing the arguments. You can access the arguments using either the name of the argument or the name specified with dest.

One useful feature of argparse is that it will convert command line arguments for you to the type specified by the type parameter. You can write your own function to do the conversion, as well.

Another feature is that argparse will automatically create a help option, -h, for your application, using the help strings provided with each option or parameter.

argparse parses the entire command line, not just arguments

Table 13. add_argument() named parameters

parameter	description
dest	Name of attribute (defaults to argument name)
nargs	Number of arguments Default: one argument, returns string '*': 0 or more arguments, returns list '+': 1 or more arguments, returns list '?': 0 or 1 arguments, returns list N: exactly N arguments, returns list
const	Value for options that do not take a user-specified value
default	Value if option not specified
type	type which the command-line arguments should be converted ; one of 'string', 'int', 'float', 'complex' or a function that accepts a single string argument and returns the desired object. (Default: 'string')
choices	A list of valid choices for the option
required	Set to true for required options
metavar	A name to use in the help string (default: same as dest)
help	Help text for option or argument

Example

parsing_args.py

```
#!/usr/bin/env python
import re
import fileinput
import argparse
from glob import glob  ①
from itertools import chain  ②

arg_parser = argparse.ArgumentParser(description="Emulate grep with python")  ③

arg_parser.add_argument(
    '-i',
    dest='ignore_case', action='store_true',
    help='ignore case'
)  ④

arg_parser.add_argument(
    'pattern', help='Pattern to find (required)'
)  ⑤

arg_parser.add_argument(
    'filenames', nargs='*',
    help='filename(s) (if no files specified, read STDIN)'
)  ⑥

args = arg_parser.parse_args()  ⑦

print('-' * 40)
print(args)
print('-' * 40)

regex = re.compile(args.pattern, re.I if args.ignore_case else 0)  ⑧

filename_gen = (glob(f) for f in args.filenames)  ⑨
filenames = chain.from_iterable(filename_gen)  ⑩

for line in fileinput.input(filenames):  ⑪
    if regex.search(line):
        print(line.rstrip())
```

① needed on Windows to parse filename wildcards

② needed on Windows to flatten list of filename lists

③ create argument parser

④ add option to the parser; dest is name of option attribute

⑤ add required argument to the parser

⑥ add optional arguments to the parser

⑦ actually parse the arguments

⑧ compile the pattern for searching; set re.IGNORECASE if -i option

⑨ for each filename argument, expand any wildcards; this returns list of lists

⑩ flatten list of lists into a single list of files to process (note: both filename_gen and filenames are generators; these two lines are only needed on Windows — non-Windows systems automatically expand wildcards)

⑪ loop over list of file names and read them one line at a time

parsing_args.py

```
usage: parsing_args.py [-h] [-i] pattern [filenames [filenames ...]]
parsing_args.py: error: the following arguments are required: pattern, filenames
```

parsing_args.py -i '\bbil' ../DATA/alice.txt ../DATA/presidents.txt

```
----------------------------------------
Namespace(filenames=['../DATA/alice.txt', '../DATA/presidents.txt'], ignore_case=True,
pattern='\\bbil')
----------------------------------------
                The Rabbit Sends in a Little Bill
Bill's got the other--Bill! fetch it here, lad!--Here, put 'em up
Here, Bill! catch hold of this rope--Will the roof bear?--Mind
crash)--`Now, who did that?--It was Bill, I fancy--Who's to go
then!--Bill's to go down--Here, Bill! the master says you're to
  `Oh! So Bill's got to come down the chimney, has he?' said
Alice to herself.  `Shy, they seem to put everything upon Bill!
I wouldn't be in Bill's place for a good deal:  this fireplace is
above her:  then, saying to herself `This is Bill,' she gave one
Bill!' then the Rabbit's voice along--`Catch him, you by the
  Last came a little feeble, squeaking voice, (`That's Bill,'
The poor little Lizard, Bill, was in the middle, being held up by
end of the bill, "French, music, AND WASHING--extra."'
Bill, the Lizard) could not make out at all what had become of
Lizard as she spoke.  (The unfortunate little Bill had left off
42:Clinton:William Jefferson 'Bill':1946-08-19:NONE:Hope:Arkansas:1993-01-20:2001-01-
20:Democratic
```


parsing_args.py -h

```
usage: parsing_args.py [-h] [-i] pattern [filenames [filenames ...]]

Emulate grep with python

positional arguments:
  pattern     Pattern to find (required)
  filenames   filename(s) (if no files specified, read STDIN)

optional arguments:
  -h, --help  show this help message and exit
  -i          ignore case
```

Simple Logging

- Specify file name
- Configure the minimum logging level
- Messages added at different levels
- Call methods on logging

For simple logging, just configure the log file name and minimum logging level with the basicConfig() method. Then call one of the per-level methods, such as logging.debug or logging.error, to output a log message for that level. If the message is at or above the minimal level, it will be added to the log file.

The file will continue to grow, and must be manually removed or truncated. If the file does not exist, it will be created.

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

Table 14. Logging Levels

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

Example

logging_simple.py

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

import logging

logging.basicConfig(
    filename='../TEMP/simple.log',
    level=logging.WARNING,
) ①

logging.warning('This is a warning') ②
logging.debug('This message is for debugging') ③
logging.error('This is an ERROR') ④
logging.critical('This is ***CRITICAL***') ⑤
logging.info('The capital of North Dakota is Bismark') ⑥
```

① setup logging; minimal level is WARN

② message will be output

③ message will NOT be output

④ message will be output

⑤ message will be output

⑥ message will not be output

simple.log

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

Formatting log entries

- Add format=format to basicConfig() parameters
- Format is a string containing directives and (optionally) other text
- Use directives in the form of %(item)type
- Other text is left as-is

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

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

Example

logging_formatted.py

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

import logging

logging.basicConfig(
    format='%(name)s %(asctime)s %(levelname)s %(message)s', ①
    filename='../TEMP/formatted.log',
    level=logging.INFO,
)

logging.info("this is information")
logging.warning("this is a warning")
logging.info("this is information")
logging.critical("this is critical")
```

① set the format for log entries

formatted.log

```
root 2021-01-21 16:17:17,803 INFO this is information
root 2021-01-21 16:17:17,803 WARNING this is a warning
root 2021-01-21 16:17:17,803 INFO this is information
root 2021-01-21 16:17:17,803 CRITICAL this is critical
```


Table 15. Log entry formatting directives

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

Logging exception information

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

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

Example

logging_exception.py

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

import logging

logging.basicConfig( ①
    filename='../TEMP/exception.log',
    level=logging.WARNING,  ②
)

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

① configure logging

② minimum level

③ add exception info to the log

exception.log

```
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "logging_exception.py", line 12, in <module>
    result = i/0
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "logging_exception.py", line 12, in <module>
    result = i/0
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "logging_exception.py", line 12, in <module>
    result = i/0
ZeroDivisionError: division by zero
```

Logging to other destinations

- Use specialized handlers to write to other destinations
- Multiple handlers can be added to one logger
  - NTEventLogHandler for Windows event log
  - SysLogHandler for syslog
  - SMTPHandler for logging via email

The logging module provides some preconfigured log handlers to send log messages to destinations other than a file.

Each handler has custom configuration appropriate to the destination. Multiple handlers can be added to the same logger, so a log message will go to a file and to email, for instance, and each handler can have its own minimum level. Thus, all messages could go to the message file, but only CRITICAL messages would go to email.

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

NOTE On Windows, you must run the example script (logging.altdest.py) as administrator. You can find `Command Prompt (admin)` on the main Windows 8/10 menu. You can also right-click on `Command Prompt` from the Windows 7 menu and choose "Run as administrator".

Example

logging_altdest.py

```
#!/usr/bin/env python
import sys
import logging
import logging.handlers

logger = logging.getLogger('ThisApplication')  ①
logger.setLevel(logging.DEBUG)  ②

if sys.platform == 'win32':
    eventlog_handler = logging.handlers.NTEventLogHandler("Python Log Test")  ③
    logger.addHandler(eventlog_handler)  ④
else:
    syslog_handler = logging.handlers.SysLogHandler()  ⑤
    logger.addHandler(syslog_handler)  ⑥

# note -- use your own SMTP server...
email_handler = logging.handlers.SMTPHandler(
    ('smtpcorp.com', 8025),
    'LOGGER@pythonclass.com',
    ['jstrick@mindspring.com'],
    'ThisApplication Log Entry',
    ('jstrickpython', 'python(monty)'),
)  ⑦

logger.addHandler(email_handler)  ⑧

logger.debug('this is debug')  ⑨
logger.critical('this is critical')  ⑨
logger.warning('this is a warning')  ⑨
```

① get logger for application

② minimum log level

③ create NT event log handler

④ install NT event handler

⑤ create syslog handler

⑥ install syslog handler

⑦ create email handler

⑧ install email handler

⑨ goes to all handlers

Chapter 5 Exercises

Exercise 5-1 (copy_files.py)

Write a script to find all text files (only the files that end in ".txt") in the DATA folder of the student files and copy them to C:\TEMP (Windows) or /tmp (non-windows). On Windows, create the C:\TEMP folder if it does not already exist.

Add logging to the script, and log each filename at level INFO.

TIP	use shutil.copy() to copy the files.

Chapter 6: 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.

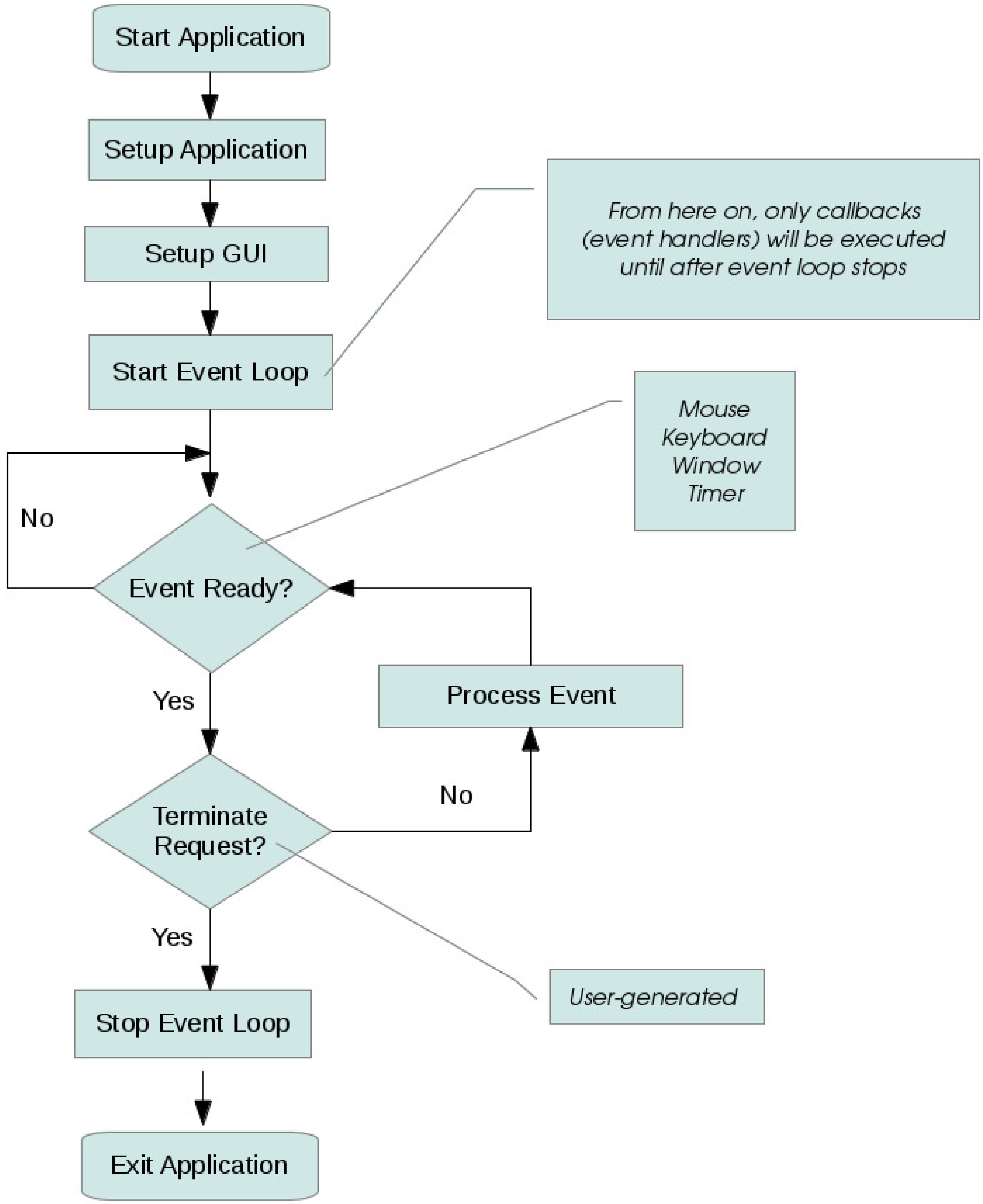
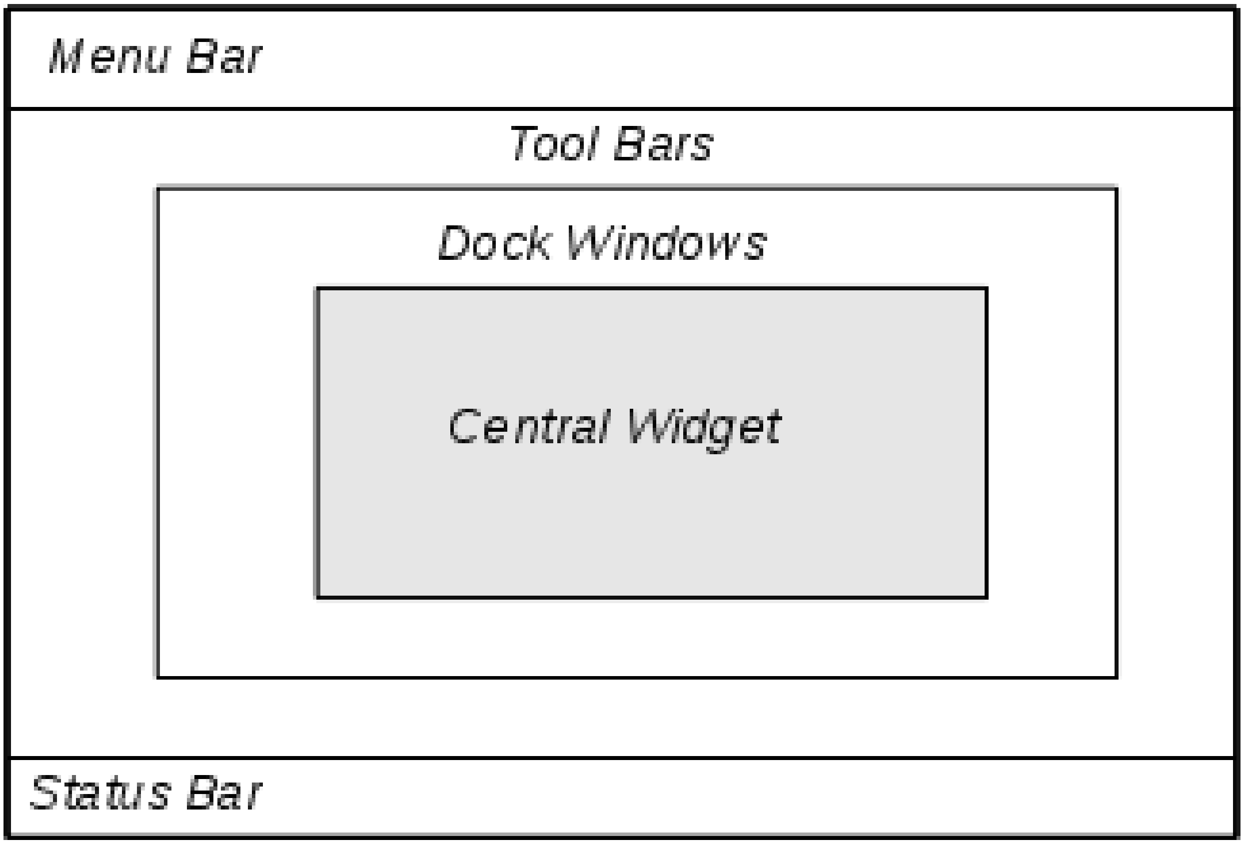


Figure 1. GUI Application Flow Chart

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.

NOTE

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.

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 16. 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

```python
#!/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()
    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 17. 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)
±jang
    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 6 Exercises

Exercise 6-1 (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.

Chapter 7: Django Overview

Objectives

- Learn what Django is, and what it can do
- See what files are generated by Django
- Understand the difference between projects and apps

What is Django?

- Full-featured web framework
- Many extensions
- Supports web apps and web services
- Builds basic app, you fill in details

Django is a complete framework for implementing all kinds of interactive web applications, including web services. The default install includes many subpackages to handle all aspects of web development, and there are extensions for less-common needs.

Django provides the tools and components to rapidly create a web app, storing data in any popular database, and using templates to generate HTML. An admin interface to your database is included in the basic framework.

Being a framework, Django creates the fundamental project structure, with Python scripts ready to fill in with your details. Configuration is handled by several predefined scripts.

The rationale for Django is to handle the tedious parts of Web development, such as DB admin, state, sessions, security, etc., and let the developers focus on the domain-specific part of their apps.

Developers should not have to reinvent any wheels when using Django. Django scales easily from a tiny app to global commercial websites.

Django features

- Models (Object Relational Mapper)
- Views (functions that render templates or data)
- Controllers (implicit in Django architecture)
- Web server for testing
- Form handling
- Unit testing
- Caching framework
- I18N
- Serialization (very handy for web services!)
- Admin interface
- Extensible authentication system
- Support for individual sites sharing apps
- Protection from XSS, SQL injection, password cracking, etc.

The motto from Django's home page is "The Web framework for perfectionists with deadlines"

Who created Django?

- Created at Lawrence World-Journal
- Named for Django Reinhardt
- First developed 2003
- Released publicly 2005
- Released to Django Software Foundation 2008

Django was created in 2003 by programmers at the Lawrence World-Journal newspaper. They had become frustrated with PHP, and had their own ideas about how a web framework should be implemented.

Django was released publicly under the BSD software license in 2005. It has quickly grown in popularity.

Django is now maintained by the non-profit Django Software Foundation.

The name comes from Django Reinhardt, a well-known jazz guitarist who was active in the Paris jazz scene in the 1930s and 1940s.

Django in a nutshell

- Create project and app
- Build model(s)
- Define views
- Design templates
- Map URLs

It's easy to get started with Django. The first step is to generate a project and an app.

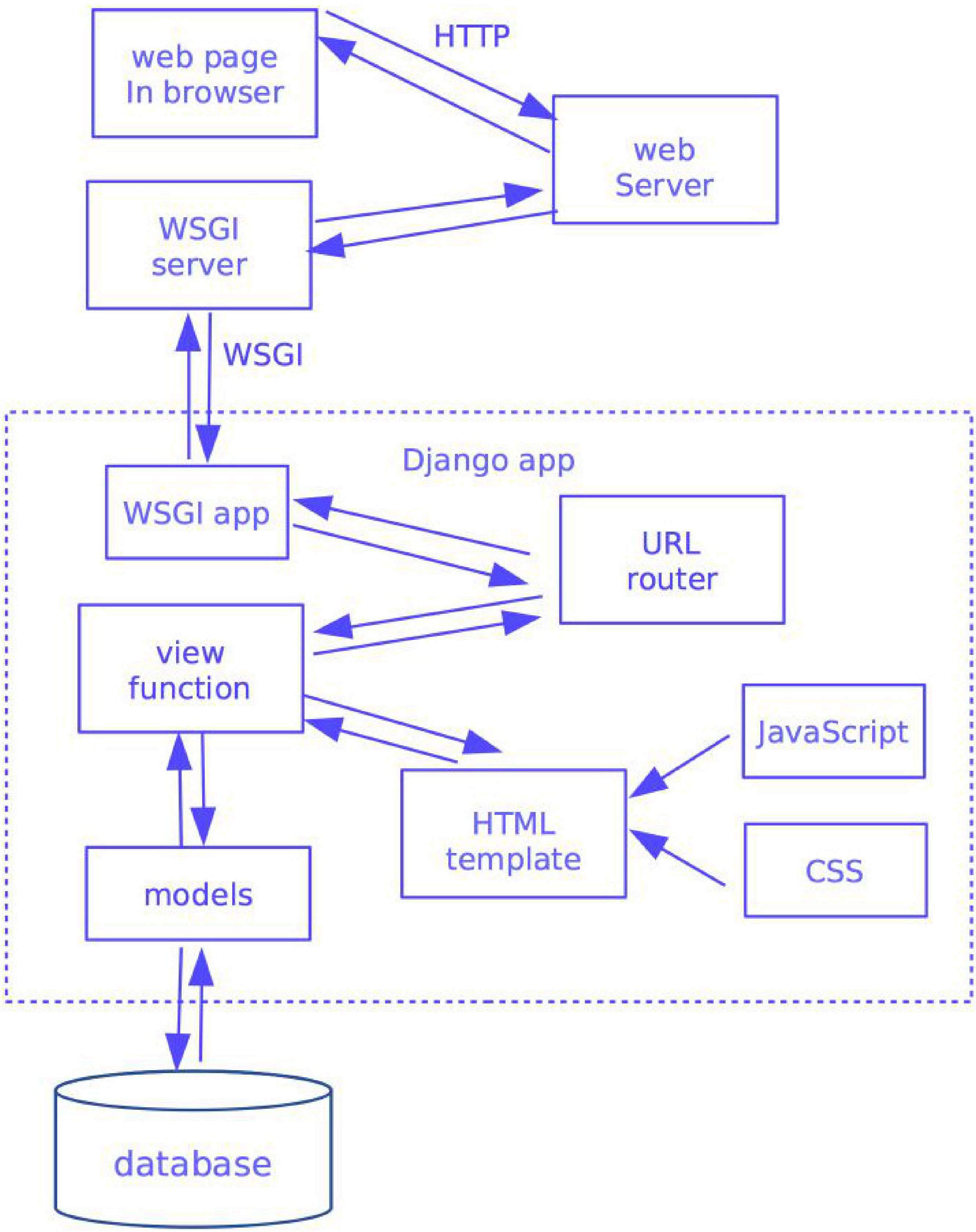
Assuming you will keep permanent data as part of the application, the next step is to define *models*. These map to tables in the database. The database can be new or existing. If it's existing, then Django can generate models from the database; if it's new, then you can define the models in Python and Django will generate the needed SQL to create the DB.

The next step is to create some *view functions*. A view function retrieves data via your models and passes the data to a template renderer, which returns the final web page that gets sent back to the browser.

Templates are files that contain the source to a web page – HTML, CSS, and JavaScript – as well as placeholders for app data. A view function sends data to a template renderer, which fills in the placeholders from the data and returns the final version of the web page.

The last piece of the puzzle is to map URLs in your app to specific view functions. This is done at both the project and app level.

Django Architecture



Projects and apps

- Projects are a collection of apps
- Apps are a collection of pages

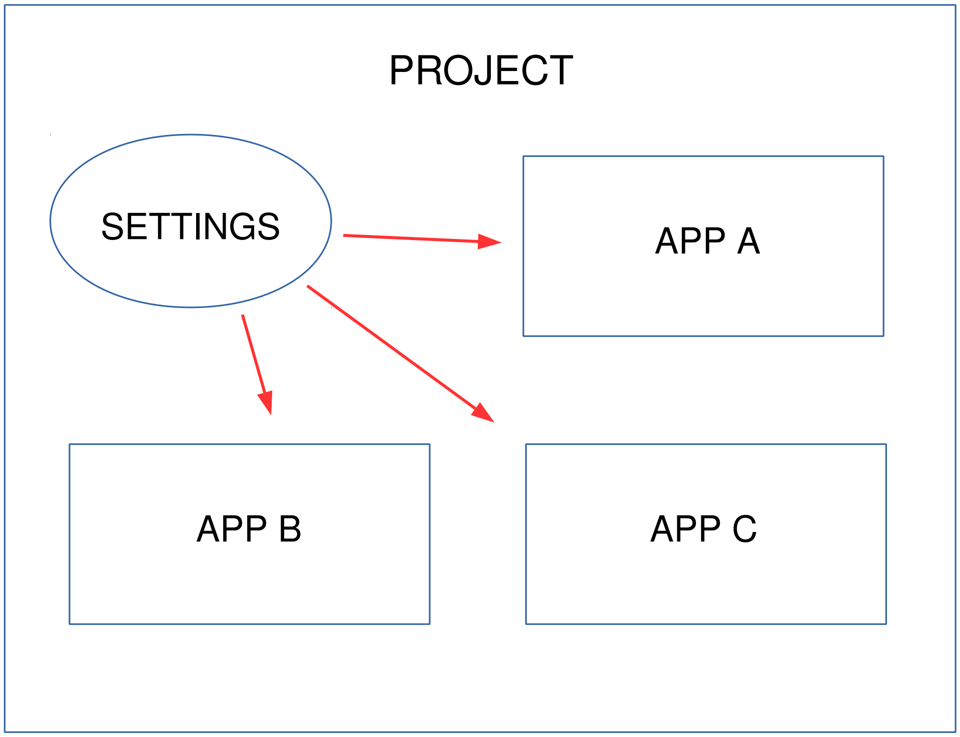
In Django, the first step is to create a project, and then create one or more apps within the project. While you can set up your project differently, it makes sense to work with Django's defaults.

An app is a Python package containing related modules, including models, views, url config, and templates.

A project is a collection of apps that shares the same configuration. This usually means the same database, among other things. While one project may be associated with one web site, Django setup is really very flexible.

All Django requires is that there be a project, and at least one app. Once you learn the ropes, you can even break that rule. The same app can be used in multiple projects.

When you create a project with **django-admin**, it creates some default folders and modules. Then, you can create one or more apps within the project.


PROJECT
SETTINGS
APP A
APP B
APP C

Chapter 8: Getting Started

Objectives

- Learn what a Django site contains
- See example Django configuration
- Build a minimal Django application

What is a site?

- One instance of Django
- Collection of apps
- Shared configuration

A **site** is one instance of Django, running on a specific host (technically, a specific IP address) and on a specific port. One site can contain any number of web apps (apps). A project contains the site, which has shared configuration that all the apps will use. We can refer to a Django **project** as one *programming project*—a folder containing a site, some apps, and other components such as documentation. The project folder contains **manage.py**, a script for managing the site.

This shared configuration includes URL routing, database configuration, middleware, and other settings.

NOTE Some developers use **site** and **project** interchangeably.

Starting a project

Django has a minimum set of files needed for a site. While you *can* write a complete Django app in one file (see **minimalapp.py** in EXAMPLES), this is not recommended. It's easier to maintain a project when code is separated into functional areas. Thus, URL configuration goes in one module, views go in another module, and so on.

While you can create a project manually, most developers use a startup tool. This will create a set of files to get the site started.

The builtin script **django-admin** has a command **startproject**, which will create a simple site layout:

```
django-admin startproject projectname
```

This creates the needed files, including **settings.py** with default configuration values.

If you created a site named **helloworld**, the project files would look similar to this:

```
helloworld              # project folder
 ├──── helloworld          # site package
 │     ├──── __init__.py
 │     ├──── settings.py
 │     ├──── urls.py
 │     └──── wsgi.py
 └──── manage.py           # site manager
```

Note that there is a package with the same name as the site under the top-level folder.

NOTE If django-admin is not in your path, try `python -m django startproject`.

We'll look at **cookiecutter**, a more thorough startup tool, in a later chapter.

manage.py

- Application manager script
- Manages database, server, etc.

manage.py is an admin script created for each app. It is similar to django-admin, but is customized for your site.

It is used to create apps, to manage databases, and to run the development server, among other things.

There are many subcommands. We will not necessarily use all possible commands in this course, but we will use many of them.

The general syntax is

```
python manage.py subcommand [param] ...
```

To get a list of subcommands, use

```
python manage.py help
```

Once you have created a site, you can run the development server immediately. Go to the folder with manage.py, and type

```
python manage.py runserver
```

Go to localhost:8000 in a browser and you should see a default page.

manage.py subcommands

auth changepassword createsuperuser **contenttypes** remove_stale_contenttypes **django** check compilemessages createcachetable dbshell diffsettings dumpdata flush inspectdb loaddata makemessages makemigrations migrate sendtestemail shell showmigrations sqlflush sqlmigrate sqlsequencereset squashmigrations startapp startproject test testserver	**sessions** clearsessions **staticfiles** collectstatic findstatic runserver

Shared configuration

- Stored in site/site/settings.py
- Installed apps (yours + plugins)
- URL mappings
- Database info
- Password validators
- Time zone info
- Location of static files

The settings.py script in the site module contains overall site configuration.

Among its contents are the installed apps for the site, which includes your apps, as well as any plugin apps provided by Django. There are several such apps provided in the default installation.

There is also top-level configuration for the URLs on your site (from site/site/urls.py). However, URLs are normally configured within each app.

This is where (by default) you specify what database (or databases) the apps in the site will use. In addition, you can specify password validators, time zone info, the location of static files, and many other details.

The settings module does not have to be called settings.py, but it is a convention that many developers follow.

NOTE See all possible settings here: https://docs.djangoproject.com/en/1.9/ref/settings/

Example

```
django-admin startproject zoo
cd zoo
python manage.py startapp wombat
python manage.py startapp koala
```

produces

```
zoo                           # project
 ├──── koala                     # app
 │     ├──── __init__.py
 │     ├──── admin.py
 │     ├──── apps.py
 │     ├──── migrations
 │     │     └──── __init__.py
 │     ├──── models.py
 │     ├──── tests.py
 │     └──── views.py
 ├──── manage.py                 # site manager
 ├──── wombat                    # app
 │     ├──── __init__.py
 │     ├──── admin.py
 │     ├──── apps.py
 │     ├──── migrations
 │     │     └──── __init__.py
 │     ├──── models.py
 │     ├──── tests.py
 │     └──── views.py
 └──── zoo                       # site
     ├──── __init__.py
     ├──── settings.py
     ├──── urls.py
     └──── wsgi.py
```

Steps to create a Django App

- Start app with manage.py (or other tool)
- Create database models
- Design templates
- Implement views
- Add app to settings
- Map urls
- Deploy app

To create a Django app, first create the app using **manage.py startapp**. This will create a minimal app package. You will add modules to it as needed.

Once the app is created, add the app name to **INSTALLED_APPS** in the settings module (default **settings.py**).

The next step is to create one or more *models* in **models**. This is optional if you aren't going to be using Django to access a database. In this chapter we'll skip models for now.

After the models are created, you can create a template. A template is an HTML file with placeholders for your app's data. There is no particular rule for what the templates contain, but typically they display either data for the user to see, or else forms for the user to fill out. We will skip templates for the very simple projects in this chapter

Now it's time to create one or more views in the **views** module, to provide data for the templates. A view is called when a particular URL is requested by the client (browser). The view connects the data to the template, and returns an HTTPResponse with the filled-out template (or other content).

To make views accessible, they must be matched to a particular URL within your application. This is done by adding entries to **urlpatterns** in the **urls** module.

Once the app has been created with startapp, you can do the above steps in any order.

The final step is to deploy the app on a web server. For development, we will use Django's builtin server.

NOTE You can add as many apps to a site as you like.

Creating the app

- Command:

```
manage.py startapp appname
```

- Creates folder (package) for app
- Provides empty modules

To create the app, go to the top level of the site. Issue the command

```
python manage.py startapp appname
```

This will create a folder named *appname*. In it, you will find some default app modules. These are not necessarily all the modules needed, and you don't have to keep those names; however, Django has a lot of "configuration by convention", so it's a good idea not to change names unless you have a good reason.

The app and the site cannot have the same name, because Django automatically creates a module in the site with the same name as the site. This is where the site-wide configuration will be stored. Keep the names simple, as they will also be Python module names that you will use in your code. Names must follow the normal rules of Python identifiers – letters, digits, and underscores. They cannot start with a digit, and most developers avoid underscores in site and app names.

Table 18. Default App Modules and Packages

admin.py	general admin settings
apps.py	application definition (defaults are OK)
migrations	folder for DB migration info
models.py	DB definitions
tests.py	unit tests
views.py	view functions

Register the app

- app object automatically created
- Register in site’s settings.py

A minor, but crucial, step is register the app in the site’s settings.py module.

Add the app name (as a string) to the **INSTALLED_APPS** list in the site-level settings.py that was auto-generated.

NOTE When using the "official" cookiecutter template **cookiecutter-django**, add the app name to **LOCAL_APPS** in **config/base.py**.

Example

django2/demo/demo/settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'greet',
]
```

TIP Everyone forgets to do this, so do it as soon as you create the app.

Create views

- Return content to browser
- Just normal functions
- Expect request parameter
- Return HTML

The next step is to create one or more view functions. These functions return some content to the browser, typically HTML.

The view functions are just normal Python functions. They are passed the HTTP request object, from which the app can get URL parameters, and other information.

In later chapters, we will render HTML templates, but for simplicity right now we will use the **HttpResponse** class provided by Django.

An HttpResponse takes a string to return, and automatically adds a default HTTP status code and default HTTP headers.

NOTE Other HTTP status codes and headers can be specified by adding parameters to HttpResponse.

Example

django2/demo/greet/views.py

```
from django.http import HttpResponse

def home(request):
    return HttpResponse("Hello, Django World")
```

The request object

- Passed into all views
- Contains information from the HTTP request

All views are passed an **HttpRequest** object as a parameter. This represents in the incoming HTTP request. From this object you can get any of the details about the request.

Some of the information available:

- Original full path of the request
- HTTP headers
- Browser ID string
- Cookies
- User name
- Browser IP

Configure the URLs

- Associate URLs with view functions
- Create urls.py in app
- Configure in site's urls.py

To associate views with particular paths (URLs), you create a url mapper, typically in a file called **urls.py**.

You can map URLs at both the site level (***site***/***site***/**urls.py**) or the app level(***site***/***app***/**urls.py**).

In either case, urls.py contains a list named **urlpatterns** that contains one or more URL mappings.

In Django 1.x, URL mappings will be **url** objects. These objects are initialized with a regular expression, the view function itself, and a label (the "view name") that refers to that particular path. This view name can be used to automatically construct a URL that points to the view from other locations.

In Django 2.x, URL mappings will be **path** objects. They work the same way, but the first argument is just a string, not a regular expression.

The view functions must be imported into urls.py.

Example

django2/demo/demo/urls.py

```
"""demo URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/1.11/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  url(r'^$', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  url(r'^$', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.conf.urls import url, include
    2. Add a URL to urlpatterns:  url(r'^blog/', include('blog.urls'))
"""
from django.conf.urls import url
from django.contrib import admin
from greet.views import home

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^$', home)
]
```

The development server

- Minimal web server
- Do not use for production
- Single-threaded
- Extra debugging support

For convenience while developing apps, Django provides a minimal web server that is integrated into your project. This server does not support multiple clients, and while it is reasonably fast, it is not fast enough for real-world services.

To start the server, you should be in the top-level folder of your project (the one that contains **manage.py**. Type

```
python manage.py runserver
```

It does contain extra debugging support. The manage.py tool described next is used to start the development server.

You should definitely not use the builtin server for production, due to the above and other factors. It is not secure enough for the real world. For production, use NGINX, Apache, IIS, or other proven servers.

Serve app with builtin server

- Use **manage.py runserver**
- http://localhost:8000 (default)

All that's left is to deploy the app. We'll discuss later how to deploy in production, but for development you can use Django's builtin server.

To launch the app, just say

```
python manage.py runserver
```

That's all there is to it.

Chapter 8 Exercises

Exercise 8-1 projectone

Using Django's builtin project starter (django-admin), create a new Django project named **djangohello**. Add an app named **hello**. Follow the steps outlined in this chapter. Create a view named **home** which returns the phrase "I am now a Django developer" as an **HTTPResponse.**

Configure the project's urls.py to map the empty URL to the **home** view.

Use **manage.py** to start up the site. Go to a browser, type in **http://localhost:8000**, and view your handiwork.

Chapter 9: Login for nothing and admin for free

Objectives

- Understand what the admin interface is for
- Implement admin for one or more apps
- Add and modify data via the admin interface

The admin Interface

- Admin for site users and groups
- Data entry and update for app data

Django provides an admin interface that allows you manage the users of your site (project). It also provides a simple interface to your models, which lets you perform CRUD functions from a browser.

To set up the admin interface, you need a superuser name and password, and you need to configure the admin interface for your apps.

CAUTION The admin interface is intended for internal use; it it not designed to be a front end for your app.

Setting up the admin user

- Need an administrative user/password
- Use python manage.py createsuperuser
- Password must be >= 8 chars

Before you can set up the admin interface, you need to create the superuser – the admin for your site. This is done with the **createsuperuser** command to manage.py:

```
python manage.py createsuperuser
```

It will prompt you for name, email, and password.

```
$ python manage.py createsuperuser
Username (leave blank to use 'jstrick'): sitemgr
Email address: sitemgr@mysite.com
Password:
Password (again):
Superuser created successfully.
$
```

Configuring admin for your apps

- Register models with admin
- Usually in admin.py

To use the admin interface with your models, you need to register them. All that's needed is to edit the admin.py module in your app. Import the models you want to access with admin, and then call admin.site.register()

Example

django2/djmodels/superheroes/admin.py

```
from django.contrib import admin

# Register your models here.

from superheroes.models import Superhero, Power, Enemy, City ①

admin.site.register(Superhero)  ②
admin.site.register(City) ②
admin.site.register(Enemy) ②
admin.site.register(Power) ②
```


Using the admin interface

- Start development server
- Go to host:port/admin
- Log in as superuser

To use the admin interface, just start the development server and go to /admin. You'll need to log in with the superuser name and password, and then you will see your models listed. Click on any model to edit.

Tweaking the admin interface

- Create Admin model
- Add options and validations

By default, you can just register models with the admin interface, and they will work. However, for special cases you may want to create Admin models, which map to the actual models, and register the Admin models. These allow you to add options and validation to the admin interface for particular models. To create an Admin model, define a class that inherits from admin.ModelAdmin. Register that class by calling admin.site.register(). The first parameter is the model, the second is the model admin class.

Example

django2/djmodels/superheroes/admin2.py

```
from django.contrib import admin

# Register your models here.

from superheroes.models import Superhero, Power, Enemy, City ①

admin.site.register(Superhero)  ②
admin.site.register(City) ②
admin.site.register(Enemy) ②
admin.site.register(Power) ②


class PowerAdmin(admin.ModelAdmin):  ③
    search_fields = ['name', 'description'] ④

admin.site.register(Power, PowerAdmin) ⑤
```

See https://docs.djangoproject.com/en/1.10/ref/contrib/admin/ for more details on Admin modules

Table 19. ModelAdmin Options

Option	Description
ModelAdmin.actions	List of actions available on change list page
ModelAdmin.actions_on_top	Controls where actions bar appears
ModelAdmin.actions_on_bottom	Same…
ModelAdmin.actions_selection_counter	Whether selection counter displayed next to the action
ModelAdmin.date_hierarchy	Set date_hierarchy to name of DateField or DateTimeField
ModelAdmin.empty_value_display	Override default display value for empty fields
ModelAdmin.exclude	List of field names to exclude from form.
ModelAdmin.fields	List of fields to show
ModelAdmin.fieldsets	Set fieldsets to control layout of add/change pages
ModelAdmin.filter_horizontal	Use JavaScript “filter” interface for options
ModelAdmin.filter_vertical	Same as filter_horizontal, but vertical
ModelAdmin.form	Specify custom form for admin pages
ModelAdmin.formfield_overrides	Quick-and-dirty override of Field options
ModelAdmin.inlines	Add inline editing of related fields
ModelAdmin.list_display	Set which fields are displayed change list page
ModelAdmin.list_display_links	If and which fields in list_display linked to “change” page
ModelAdmin.list_editable	List of field names allowing editing on change list page
ModelAdmin.list_filter	Activate filters in right sidebar of change list page
ModelAdmin.list_max_show_all	Max items on “Show all” admin change list page
ModelAdmin.list_per_page	Max items on paginated list page
ModelAdmin.list_select_related	Use select_related() to retrieve objects on admin change list page
ModelAdmin.ordering	Specify how lists of objects should be ordered in admin views
ModelAdmin.paginator	Which paginator class is used for pagination
ModelAdmin.prepopulated_fields	Map field names to the values for prepopulation
ModelAdmin.preserve_filters	Preserve filters on list view

Option	Description
ModelAdmin.radio_fields	Use radio-buttons for foreign keys rather than select
ModelAdmin.raw_id_fields	List of static values for foreign keys
ModelAdmin.readonly_fields	List of fields to be non-editable
ModelAdmin.save_as_continue	Redirect to change view, otherwise changelist view
ModelAdmin.save_on_top	Add save buttons across top of admin change forms
ModelAdmin.search_fields	Enable search box on admin change list page
ModelAdmin.show_full_result_count	Control display of full count of objects
ModelAdmin.view_on_site	Whether or not to display "View on site"

Chapter 9 Exercises

Exercise 9-1 (music)

Set up the admin interface for the music project. Using the admin interface, add some more bands to your database.

Chapter 10: Creating models

Objectives

- Understand ORM
- Create and populate models
- Initialize the database
- Access data through the ORM

What is ORM?

- Object Relational Mapper
- Maps objects (Python classes) to DB tables
- No SQL needed
- Trades convenience for overhead

ORM stands for Object Relational Mapper. It refers to creating classes (in any language) that map to database tables. ORMs in other languages include Hibernate, NHibernate, and Spring.

Django has its own builtin ORM.

All data manipulation is done via the models; no SQL ever needs to be written. Behind the scenes, SQL is generated by Django routines. ORM adds a little overhead, but makes working with data very convenient.

For the rare situation that is not covered by ORM, Django provides a way to execute raw SQL.

> The most popular Python ORM outside of Django is SQLAlchemy.

Defining models

- Python class <⇒ DBMS table
- Can specify constraints (foreign keys, etc.)
- Inherit from django.models

If you are starting your database from scratch, you will have to define models in terms of Django Model objects. These will ultimately be translated into SQL tables.

Every model needs a unique ID.

For each "table", define a class that inherits from django.db.models.Model. Within each class, define a field (DB column) as a class variable, assigned from one of the Django model field types (see next page for list).

You can create foreign keys and other relations with special field types. The most common special field types are ForeignKey and ManyToManyType.

For each model, add a *str*\(\) method. This should return a string that represents the model. Typically it will return the name field of the object, but it can return anything that makes sense to the developer, as long it it’s a string.

Example

django2/djmodels/superheroes/models.py

```python
from django.db import models

class Power(models.Model):
    name = models.CharField(max_length=32, unique=True)
    description = models.CharField(max_length=512)

    def __str__(self):
        return self.name

class City(models.Model):
    name = models.CharField(max_length=32, unique=True)

    def __str__(self):
        return self.name

class Enemy(models.Model):
    name = models.CharField(max_length=32, unique=True)
    powers = models.ManyToManyField(Power)

    def __str__(self):
        return self.name

class Superhero(models.Model):
    name = models.CharField(max_length=32, unique=True)
    real_name = models.CharField(max_length=32)
    secret_identity = models.CharField(max_length=32)
    city = models.ForeignKey(City, on_delete=models.CASCADE)
    powers = models.ManyToManyField(Power)
    enemies = models.ManyToManyField(Enemy)

    def __str__(self):
        return self.name
```


Table 20. Django Model Field Types

Data fields	Relationship fields
AutoField BigAutoField BigIntegerField BinaryField BooleanField CharField CommaSeparatedIntegerField DateField DateTimeField DecimalField DurationField EmailField FileField FileField and FieldFile FilePathField FloatField ImageField IntegerField GenericIPAddressField NullBooleanField PositiveIntegerField PositiveSmallIntegerField SlugField SmallIntegerField TextField TimeField URLField UUIDField	ForeignKey ManyToManyField OneToOneField

Relationship fields

- Fields that relate to other models
- Three special field types
  - Foreign Key one-to-many
  - ManyToManyField many-to-many
  - OneToOneField one-to-one

The reason database systems are called relational is that tables (and in Django, models) are related to each other. There are several types of relationships.

A *foreign key relation* is one-to-many. An entry in one table contains the ID of an entry in another table, called the child table. Either end can be the "main" or "most important" table. A common example of this is customers and orders. In this scenario, each Order model has a Customer ID field that links it to a particular customer. In the Order model, Customer ID is a foreign key. In the Django ORM, you specify the name of the foreign model with a ForeignKey field.

A *many-to-many* relation occurs when you have two tables that might refer to each other. For instance, A book might have several authors, and an author might have several books. This is achieved in the database by having a third table that maps the association between the other two tables. The Django ORM creates this table automatically when you use a ManyToManyField. From one of the models, you specify the other side of the relationship in the ManyToManyField. Only do this on one side. That is, don’t use a ManyToManyField on both models. Either one is fine. The ORM will do the right thing.

A *one-to-one* relation is similar to a foreign key relation, except that there can only be one related object in the related model. It is implemented with the OneToOneField. Practically speaking, a model might have a list of values from its foreign key, but only one from a OneToOneField.

Creating and migrating models

- Managed by Django
- Keeps track of changes to database schema
- Allows reversion to earlier schema
- Like git for your database

Django will manage and keep track of changes to your database schema via *migrations*. Each update is numbered, and you can revert to previous versions.

A later chapter will cover data migration in detail.

To actually create the tables, run

```
python manage.py makemigrations
```

followed by

```
python manage.py migrate
```

This will execute the SQL to actually create the tables. Each time you change anything in your models, re-run the above commands.

Be sure the database (which can be empty) exists before running any Django commands

Using existing tables

- Django will build the models
- Use manage.py inspectdb > models.py
- Will model entire database – deleted unneeded tables

If you already have a database and your tables defined, you don't have to write the models yourself.

Use

```
python manage.py inspectdb
```

to generate the models for the entire database. This will generate models for all tables in the database, include system tables that you generally don't care about and don't want to update from your Django app.

Typically you would redirect the output to models.py, and then delete any models you didn't need. When you have narrowed models.py down to the models you actually need, then you may need to tweak and test the models until they work properly.

For instance, inspectdb sets all models to be unmanaged. This means that Django will not manage the model's lifecycle. An unmanaged model will be managed outside of your Django app. Change managed=False to managed=True in the Meta class within each model you want to have Django manage.

Once you have the models.py tweaked to your liking, use manage.py to run the makemigrations and migrate commands. Now you can access your existing tables without creating any models by hand.

TIP To use inspectdb with Oracle databases, the account needs schema owner privilege on at least dba_tables and dba_tab_cols.

Opening a Django shell

- Convenient for quick sanity checks
- Sets up Django environment
- Starts iPython (enhanced interpreter)

To interactively work with your models, you can open a shell. This opens a Python interpreter (nowadays iPython) with the project's configuration loaded.

This makes it easy to manipulate database objects.

To start the shell, type

```
python manage.py shell
```

Creating and modifying objects

- Create new object and populate
- Call save() on the object
- Equal to INSERT INTO or UPDATE

It is easy to create or update objects. Import the object from the models module, populate it as desired, and then call the save() method on the object.

You can create model instances using field names as named parameters, as well. Assigning invalid data will raise an error when save() is called.

To add foreign fields to an object, just create (or search for) the foreign object, and assign the foreign object to the appropriate field. Be sure to save the foreign object before you assign it.

To add many-to-many fields, use field.add(obj, …).

Example

```
python manage.py shell

Python 3.6.2 |Anaconda custom (x86_64)| (default, Sep 21 2017, 18:29:43)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from superheroes.models import City, Enemy, Power, Superhero

In [2]: lex = Enemy(name='Lex Luthor')

In [3]: lex.save()

In [4]: met = City(name="Metropolis")

In [5]: met.save()

In [6]: flying = Power(name="flying", description="Fly though the air unaided")

In [7]: flying.save()

In [8]: sup = Superhero(name='Superman', secret_identity='Clark Kent', real_name='Kal-
el', ci
   ...: ty=met)

In [9]: sup.save()

In [10]: sup.powers.add(flying)

In [11]: sup.enemies.add(lex)

In [12]: sup.save()

In [13]: sup.secret_identity
Out[13]: 'Clark Kent'

In [14]: ^Z
```

Accessing data

- Import models from models.py
- Use MODEL.objects to access "rows"
- Use all(), filter(), or get() to select
- Use manage.py shell for interactive work

To access data in the database, you create a QuerySet object from your model, using a Manager object. The default Manager is called, confusingly enough, objects.

A QuerySet is equivalent to the result of a SELECT statement.

Use all() to retrieve all objects.

Use filter() to narrow down the results, like using WHERE in a SQL query. Filter takes named parameters that match the fields in the model.

Use get() to retrieve a single object by a particular field.

CAUTION	get() returns one object, but all() and filter() return QuerySets, which are lists of object.

Example

```
pm shell
Python 3.6.2 |Anaconda custom (x86_64)| (default, Sep 21 2017, 18:29:43)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from superheroes.models import City, Enemy, Power, Superhero

In [2]: h = Superhero.objects.all()

In [3]: for hero in h:
   ...:     print(hero.name, hero.secret_identity)
   ...:
Superman Clark Kent
Wonder Woman Diana Prince
Spider-Man Peter Parker
Iron Man Tony Stark

In [4]: spidey = Superhero.objects.filter(name='Spider-Man').first()

In [5]: spidey.real_name
Out[5]: 'Peter Parker'

In [6]: spidey.powers.all()
Out[6]: <QuerySet [<Power: super strength>, <Power: wallclimbing>, <Power: spider-
sense>]>

In [7]: spidey.enemies.all()
Out[7]: <QuerySet [<Enemy: Doctor Octopus>, <Enemy: Green Goblin>]>

In [8]: ww = Superhero.objects.get(name='Wonder Woman')

In [9]: ww.secret_identity
Out[9]: 'Diana Prince'

In [10]:
```

NOTE We will cover detailed manipulation of data in Chapter 12: Querying Django Models

Is ORM an anti-pattern?

- ORM not required for Django apps
- Use straight SQL, NoSQL, or flat files
- Views can get data from anywhere

One thing to remember is that just because you have a hammer, don't start treating everything like a nail.

ORMs provide a convenient mapping between language objects and database objects, but there is a lot of work in maintaining them. They can also be slower than straight SQL queries, especially when you get into multiple joins on very large tables.

There is nothing that requires you to use Django's ORM in your apps. You don't even have to use a database, although most apps do need to permanently store data, and a database is usually the most convenient way to do that.

However, there are a lot of factors. A web site could read a huge flat file into memory and provide data from an in-memory data structure. Maybe the file is updated every day, and the app re-reads it after each update.

All in all, ORM works pretty well, solves 80-90% of your database issues, and is well documented and well implemented.

Here are some links discussing ORM as an anti-pattern:

https://blog.codinghorror.com/object-relational-mapping-is-the-vietnam-of-computer-science/

http://blogs.tedneward.com/post/the-vietnam-of-computer-science/

http://martinfowler.com/bliki/OrmHate.html

http://seldo.com/weblog/2011/08/11/orm_is_an_antipattern

Chapter 10 Exercises

Exercise 10-1 (music)

Using the cookiecutter templates, create a new Django project named **music**, with an app named **bands**.

Define the models for the bands app. Populate the database with a few of your favorite bands.

Your models should include at least the following

```
Band
    Fields: name, genre, members
Genre:
    Fields: name
Member:
    Fields: name, bands
```

TIP In the Band model, members should be many-to-many, and genre should be a foreign key.

Chapter 11: Creating Views

Objectives

- Learn what a view is for
- Translate MVC to MTV
- Create views
- Return 404 pages
- Use simple templates

Views

- Function returning content to browser
- Passed incoming HTTP request
- Usually return HTML
- The "controller" part of MVC

Views are functions that return content (usually HTML) to the client (usually a browser).

While a simple view might be self-contained, most views will pull data from your models and present it using a template.

If the requested data is not available, or some kind of error occurs, you will need to return an appropriate HTTP status code.

Views are passed the request object which has information from the client, including any data or parameters to be processed by the app.

Some of the data in the request object is parsed and made available separately.

NOTE Despite being called "views", Django views are really the "controller" part of MVC as well as being some of the view logic.

Example

```
# very simple view
def home(request):
    return HTTPResponse("Welcome to my app")
```


The request object

- Passed into all views
- Contains information from the HTTP request

All views are passed an HttpRequest object as a parameter. This represents in the incoming HTTP request. From this object you can get any of the details about the request.

Some of the information available includes the original full path of the request (the complete URL), the HTTP headers, and any cookies that are on the client's browser.

Table 21. HTTP request object attributes

Attribute	Description
scheme	Scheme of the request (usually http or https)
body	The raw HTTP request (byte string)
path	Full path to requested page (not including scheme or domain)
path_info	The path portion of the URL after the host name is split into a script prefix and path info
method	HTTP method (always uppercase)
encoding	Current encoding for form submission
content_type	MIME type of the request (parsed from the CONTENT_TYPE header)
content_params	A dictionary of key/value parameters from CONTENT_TYPE header
GET	Dictionary-like object with all HTTP GET parameters
POST	Dictionary-like object with all HTTP POST parameters
COOKIES	Dictionary containing all cookies (keys and values are strings)
FILES	Dictionary-like object containing all uploaded files
META	Dictionary containing all available HTTP headers
CONTENT_LENGTH	The length of the request body (as a string).
CONTENT_TYPE	The MIME type of the request body.
HTTP_ACCEPT	Acceptable content types for the response.
HTTP_ACCEPT_ENCODING	Acceptable encodings for the response.
HTTP_ACCEPT_LANGUAGE	Acceptable languages for the response.
HTTP_HOST	The HTTP Host header sent by the client.
HTTP_REFERER	The referring page, if any.
HTTP_USER_AGENT	The client's user-agent string.
QUERY_STRING	The query string, as a single (unparsed) string.
REMOTE_ADDR	The IP address of the client.
REMOTE_HOST	The hostname of the client.
REMOTE_USER	The user authenticated by the Web server, if any.
REQUEST_METHOD	A string such as "GET" or "POST".
SERVER_NAME	The hostname of the server.
SERVER_PORT	The port of the server (as a string).

HttpRequest.resolver_match	Instance of ResolverMatch representing the resolved URL (only set after URL resolving takes place)

Route configuration

- Associates URL with view
- URL matched with regular expressions
- Two levels by default (but configurable)
  - *project*/*project*/urls.py
  - *project*/*app*/urls.py

To make views available in an app, they must be configured to match a particular URL. This is done with a URLconf.

When you create a project, it generates a default URL configuration is in project/project/urls.py. This initial config only includes the URL for the admin interface.

While you can do all the config in this file, it is usually more convenient to have app-specific config in the app. Thus, the project-level config includes a urls.py from each app.

URLs are added to a list named urlpatterns. Each pattern is a url object, which has at least two parameters. The first parameter is a regular expression that matches the URL, the second parameter is either a view function, an include(), or the as_view() method of a class-based view. django.conf.urls.include object, or a module containing a urlpatterns list.

You can add other named parameters that are passed into the view. This can be useful for customizing multiple URLs with the same view function, but makes the URL more dependent on the view.

In the project's **urls.py**, you can include URLconfs from apps. When doing this, specify a **namespace**. This can be used as a prefix when referring to URLs for that app. In the app's **urls.py**, you can include a name, which acts as a label for that URL. This makes it easy to move views around without hard-coding their paths in templates and view functions.

Example

django2/djsuper/djsuper/urls.py

```
"""djsuper URL Mapping

The `urlpatterns` list maps URLs to views. More information:
    https://docs.djangoproject.com/en/1.11/topics/http/urls/

Function views:
    1. Add an import:  from my_app import views
    2. Add entry to urlpatterns:  url(r'^$', views.home, name='home')

Class-based views:
    1. Add an import:  from my_app.views import Home
    2. Add entry to urlpatterns:  url(r'^$', Home.as_view(), name='home')

Including another (usually an app's) URLconf:
    1. Import the include() function: from django.conf.urls import url, include
    2. Add a URL to urlpatterns:  url(r'^blog/', include('blog.urls', namespace="blog"))
"""
from django.conf import settings
from django.contrib import admin

# site-wide route mapping
from django.urls import path, include
urlpatterns = [
    path('admin', admin.site.urls),
    path('', include('superheroes.urls')),
]

# include Django Debug toolbar if DEBUG is set
if settings.DEBUG:
    import debug_toolbar
    urlpatterns = [
        path('__debug__/', include(debug_toolbar.urls)),
    ] + urlpatterns
```

Example

django2/djsuper/superheroes/urls.py

```
"""
URL Configuration for superheroes
"""
from django.conf.urls import url
from . import views   # import views from app
from . import views404
from . import viewsbasictemplate
from . import viewstemplate
from . import viewsqueries

urlpatterns = [
    url('', views.home, name='home'),
    url('hero/<str:hero_name>/', views.hero, name="hero"),
    url('hero404x/<str:hero_name>/', views404.hero404, name="hero404"),
    url('hero404sc/<str:hero_name>/', views404.hero404sc, name="hero404sc"),
    url(
        'herotemplate101/<str:hero_name>/',
        viewsbasictemplate.hero_basic_template,
        name="herobasictemplate",
    ),
    url(
        'herohardway/<str:hero_name>/',
        viewstemplate.hero_hard_way,
        name="herohardway",
    ),
    url(
        'heroeasyway/<str:hero_name>/',
        viewstemplate.hero_easy_way,
        name="heroeasyway",
    ),
    url(
        'herolookups/<str:hero_name>/',
        viewstemplate.hero_lookups,
        name="herolookups",
    ),
    url(
        'herofilters/<str:hero_name>/',
        viewstemplate.hero_filters,
        name="herofilters",
    ),
    url(
        'herotags/<str:hero_name>/',
        viewstemplate.hero_tags,
```

```
            name="herotags",
        ),
        url(
            'herodetails/<str:hero_name>/',
            viewstemplate.hero_details,
            name="herodetails",
        ),
        url(
            'heroescape/<str:hero_name>/',
            viewstemplate.hero_escape,
            name="heroescape",
        ),
        url(
            'herourls/',
            viewstemplate.hero_urls,
            name="herourls",
        ),
        url(
            'herostatic/<str:hero_name>/',
            viewstemplate.hero_static,
            name="herostatic",
        ),
        url(
            'heroqueries/',
            viewsqueries.hero_queries,
            name="heroqueries",
        ),
    ]
```

Unresolved directive in ../chapters3/django_creating_views_3.asc - include::/Users/jstrick/curr/courses/python/examples3/callouts/django2/djsuper/superheroes/urls.callouts[]

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.

The following picture illustrates the the above concepts. S is a string anchor, A is an atom, W is a word anchor, and Q is a quantifier.

NOTE 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 22. 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).

HttpResponse

- Standard response for views
- Specify HTTP status code
- No HTML needed (technically)

For simple web pages, you can use the HttpResponse object provided by Django. This object takes a string of text (which is typically, but doesn't have to be, HTML), and creates a standard HTTP response, including the default status code of 200 (OK).

Example

django2/djsuper/superheroes/views.py

```
from django.http import HttpResponse
from .models import Superhero

def home(request):
    return HttpResponse("Welcome to the superhero app")

def hero(request, hero_name):
    s = Superhero.objects.get(name=hero_name)
    return HttpResponse(
        "{} is really {}".format(s.secret_identity, s.name)
    )
```

Unresolved directive in ../chapters3/django_creating_views_3.asc - include::/Users/jstrick/curr/courses/python/examples3/callouts/django2/djsuper/superheroes/views.callouts[]

Extra URL information

- Part of URL after view name
- Passed into view as extra parameters
- Use named group

It is common to pass extra information to the view via the URL.

To do this, configure the route with a fixed part (the view) and a variable part. The variable part must be enclosed in a named group. The named group syntax is **(?P<_name_>_pattern_)**.

The group name will be the name of the parameter to the view function.

For example, the URL config:

django2/djsuper/superheroes/urls.py

```
...
url(r'^hero/(?P<hero_name>.*)', views.hero, name="hero"),
...
```

Will pass everything after "hero/" into the view function as a parameter named "hero_name":

Example

django2/djsuper/superheroes/views.py

```python
from django.http import HttpResponse
from .models import Superhero

def home(request):
    return HttpResponse("Welcome to the superhero app")

def hero(request, hero_name):
    s = Superhero.objects.get(name=hero_name)
    return HttpResponse(
        "{} is really {}".format(s.secret_identity, s.name)
    )
```

Unresolved directive in ../chapters3/django_creating_views_3.asc - include::/Users/jstrick/curr/courses/python/examples3/callouts/django2/djsuper/superheroes/views.callouts[]

Django 2.0 URL Configuration

- No REs needed
- Type is specified
- REs *may* be used

404 Errors

- Page (or data) not found
- Should raise Http404 exception
- Can display custom page
- Displays error page in debug mode

The Http404 class is an error type that will generate a standard 404 error page. You can create a template with the name **404.html**, and it will automagically be used.

get_object_or_404()

- Handles common use case
- Raise 404 error if record not found
- Pass Model, Manager, or QuerySet

A very common use case in web programming is to look up a record based on user input, which is generally a query string or a URL. If the record is found, then the app should display the data; otherwise, the app should display a 404 (not found) page.

Django provides a simple shortcut for this situation, get_object_or_404(). It takes a Model, Manager, or QuerySet, plus lookup parameters (similar to model.objects.get()). It performs the specified query. If the query fails, it raises a 404 error. Otherwise, you can use the object returned.

Example

django2/djsuper/superheroes/views404.py

```
from django.http import HttpResponse
from django.shortcuts import Http404, get_object_or_404

from .models import Superhero

# Note: automagic 404 does not work when DEBUG=True

def hero404(request, hero_name):
    try:
        hero = Superhero.objects.get(name=hero_name)
    except:
        raise Http404("Hero '{}' not found [{}]".format(
                hero_name,
                request.get_full_path(),
            )
        )
    response = HttpResponse('<h1>Info for {} ({})</h1>'.format(
        hero_name,
        hero.secret_identity,
        )
    )
    return response


def hero404sc(request, hero_name):

    hero =  get_object_or_404(Superhero, name=hero_name)

    return HttpResponse('<h1>Info for {} ({})</h1>'.format(
            hero_name,
            hero.secret_identity,
        )
    )
```

Unresolved directive in ../chapters3/django_creating_views_3.asc - include::/Users/jstrick/curr/courses/python/examples3/callouts/django2/djsuper/superheroes/views404.callouts[]

About templates

- Separate presentation from data
- Get HTML, CSS, and JavaScript out of code
- Templates have placeholders for data

Even with shortcuts and writing to the HttpResponse object, there are still some issues with the views seen so far. We're still mixing the presentation (HTML, and potentially CSS and JavaScript) with the business logic. We want to be more like the MVC pattern (although Django calls it the MTV pattern – Model, Template, View).

To accomplish this, we can use templates. A template is a file containing presentation code (HTML, CSS, JavaScript), and placeholders for data. The view organizes the data from models, and passes the data and the template name to a renderer, which returns the final HTML page.

In addition to placeholders, the template can contain some directives, or simple business logic. The directives similar to, but simpler than Python itself, so they can be used by a web designer whose primary job is making the page look good, and not hard-core programming. Thus, the programmer creates the models and views, and the web designer creates the templates (whether or not they are the same person).

Using templates with views

- Get HTML, CSS, and JavaScript out of code
- Create templates folder in app
- Edit template and insert placeholders

As with all parts of Django, there is much flexibility in the use of templates.

The simplest way to use them is to go with Django's defaults. This assumes that the templates are located in a folder named templates in the app's folder.

The loader function in django.template will find and load the template, returning a template object. Then the template object's **render** method will fill out the template with the passed-in *context* dictionary and return the HTML ready to go.

Then pass the HTML to HTTPResponse as before.

NOTE In [basic_templates] you will see an easier way of finding, loading, and rendering the template.

Example

django2/djsuper/superheroes/viewsbasictemplate.py

```
from django.http import HttpResponse
from django.shortcuts import get_object_or_404, render

from .models import Superhero


def hero_basic_template(request, hero_name):
    hero =  get_object_or_404(Superhero, name=hero_name)
    context = {
        'hero_name': hero.name,
        'real_name': hero.real_name,
        'secret_identity': hero.secret_identity,
    }
    return render(request, 'hero_basic.html', context)
```


Template placeholders

- Delimited with {{ }}
- Refer to data name/value pairs passed into renderer

A template can be very complex, but the simplest template consists of normal HTML plus data placeholders of the form {{ NAME }}. When rendered, each placeholder will be replaced with the corresponding value for each NAME from a dictionary passed into the renderer. If the NAME is not in the dictionary, the placeholder is left as-is.

Example

django2/djsuper/superheroes/templates/hero_basic.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ hero_name }}</title>
</head>
<body>
<h1>{{ hero_name }}</h1>
<h2>Secret Identity: {{ secret_identity }}</h2>
<h2>Real Name: {{ real_name }}</h2>
</body>
</html>
```

Chapter 11 Exercises

Exercise 11-1 (music/bands/*)

Create a view for your band app that takes a URL like bands/band/bandname and uses a simple template to present data for the specified band.

Chapter 12: Querying Django Models

Objectives

- Understand data managers
- Learn about QuerySets
- Use database filters
- Define field lookups
- Chain filters
- Get aggregate data

Object Queries

- Model class is table, instance is row
- model.objects is a manager
- QuerySet is a collection of objects

In the Django ORM, a model class represents a table, and a model instance represents a row. You have already seen some of this in the chapter on views.

To query your data, you will usually use a Manager. The default manager is named objects, and is available from the Model class.

From the object manager, you can get all rows, filter rows, or exclude rows. You can also implement relational operations, such as greater-than or less-than.

NOTE

For this chapter, there is just one view function that uses the builtin function eval() to evaluate a list of strings containing queries, in order to avoid duplicating the queries as labels.

Access this view at http://localhost:8000/superheroes/heroqueries

Example

django2/djsuper/superheroes/viewsqueries.py

```
from django.shortcuts import get_object_or_404, render
from django.db.models import Min, Max, Count, FloatField, Q
from .models import Superhero

q_hulk = Q(name__icontains="hulk")
q_woman = Q(name__icontains="woman")

def hero_queries(request):

    queries = [
        'Superhero.objects.all()',
        'Superhero.objects.filter(name="Superman")',
        'Superhero.objects.filter(name="Superman").first()',
        'Superhero.objects.filter(name="Spider-Man").first()',
        'Superhero.objects.filter(name="Spider-Man").first().secret_identity',
        'Superhero.objects.filter(name="Superman").first().enemies.all',
        'Superhero.objects.filter(name="Spider-Man").first().powers.all',
        'Superhero.objects.filter(name="Batman").first().powers.all',
        'Superhero.objects.exclude(name="Batman")',
        'Superhero.objects.order_by("name")',
        'Superhero.objects.count()',
        'Superhero.objects.aggregate(Count("name"))',
        'Superhero.objects.aggregate(Min("name"))',
        'Superhero.objects.aggregate(Max("name"))',
        'Superhero.objects.aggregate(Min("name"),Max("name"))',
        'Superhero.objects.filter(name__contains="man").count()',
        '''Superhero.objects.filter(
                name__contains="man").exclude(name__contains="woman")''',
        '''Superhero.objects.filter(
                name__contains="man").exclude(
                name__contains="woman").count()''',
        'Superhero.objects.all()[:3]',
        'Superhero.objects.filter(name__contains="man")[:2]',
        '''Superhero.objects.filter(
            enemies__name__icontains="Luthor").first().name''',
        'Superhero.objects.filter(q_hulk | q_woman)',
    ]

    query_pairs = [
        (query, eval(query)) for query in queries
    ]
    context = {
        'page_title': 'Query Examples',
        'query_pairs': query_pairs,
    }
    return render(request, 'hero_queries.html', context)
```


Example

django2/djsuper/superheroes/templates/hero_queries.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ hero_name }}</title>
</head>
<body>
<h1>{{ hero_name }}</h1>
<h2>Secret Identity: {{ secret_identity }}</h2>
<h2>Real Name: {{ real_name }}</h2>
</body>
</html>
```

Opening a Django shell

- Convenient for quick sanity checks
- Sets up Django environment
- Starts iPython (enhanced interpreter)

To interactively work with your models, you can open a shell. This opens a Python interpreter (nowadays iPython) with the project's configuration loaded.

This makes it easy to manipulate database objects.

To start the shell, type

```
python manage.py shell
```

NOTE If iPython (recommended) is installed, the Django shell will use it instead of the builtin interpreter.

QuerySets

- Iterable collection of objects
- Roughly equivalent to "SELECT …"
- Each row contains fields
- Use manager to create

A QuerySet is a collection of objects from a model. QuerySets can have any number of filters, which control which objects are in the result set. Filters correspond to the "WHERE …" clause of a SQL query.

all(), filter(), exclude(), sortby(), and other functions return a QuerySet object. A QuerySet can itself be filtered, sorted, sliced, etc.

ORM queries are deferred (AKA lazy). The actual database query does not happen until the QuerySet is evaluated.

```
Superhero.objects.all()
[<Superhero: Superman>, <Superhero: Spiderman>, <Superhero: Batman>, <Superhero: Wonder
Woman>, <Superhero: Hulk>]

Superhero.objects.filter(name="Superman")
[<Superhero: Superman>]

Superhero.objects.filter(name="Superman").first()
Superman

Superhero.objects.filter(name="Spiderman").first()
Spiderman

Superhero.objects.filter(name="Spiderman").first().secret_identity
Peter Parker

Superhero.objects.filter(name="Superman").first().enemies.all
[<Enemy: Lex Luthor>, <Enemy: General Zod>]

Superhero.objects.filter(name="Spiderman").first().powers.all
[<Power: Super strength>, <Power: Spidey-sense>, <Power: Intellect>]

Superhero.objects.filter(name="Batman").first().powers.all
[<Power: Detective ability>]

Superhero.objects.exclude(name="Batman")
[<Superhero: Superman>, <Superhero: Spiderman>, <Superhero: Wonder Woman>, <Superhero:
Hulk>]

Superhero.objects.order_by("name")
[<Superhero: Batman>, <Superhero: Hulk>, <Superhero: Spiderman>, <Superhero: Superman>,
<Superhero: Wonder Woman>]
```


Query Functions

Returning QuerySets	**Returning Objects**
filter()	get()
exclude()	create()
annotate()	get_or_create()
order_by()	update_or_create()
reverse()	latest()
distinct()	earliest()
values()	first()
values_list()	last()
dates()	
datetimes()	**Returning other values**
none() (empty)	bulk_create() (None)
all()	count() (int)
union()	in_bulk() (dict)
intersection() (1.11+ only)	iterator() (iterator)
difference() (1.11+ only)	aggregate() (dict)
select_related() (1.11+ only)	exists() (bool)
prefetch_related()	update() (int)
extra()	delete() (int)
defer()	as_manager() (Manager)
only()	
using()	
select_for_update()	
raw()	

Field lookups

- Field comparisons
- Use *field*__operator
- Work with filter(), exclude(), distinct(), etc.

In SQL, the WHERE clause allows you to compare columns using relational operators. To do this Django, you can append special operator_suffixes to field names, such as name__greaterthan or secret_identity__contains. Note that the operators are preceded by two underscores.

Example

```
Superhero.objects.filter( name__contains="man").exclude(name__contains="woman")
[<Superhero: Superman>, <Superhero: Spiderman>, <Superhero: Batman>]

Superhero.objects.filter( name__contains="man").exclude( name__contains="woman").count()
3
```

You can also create custom lookups for model fields

Field Lookup operator suffixes

Use in **filter()**, **exclude()**, and **get()**.

```
__exact
__iexact
__contains
__icontains
__in
__gt
__gte
__lt
__lte
__startswith
__istartswith
__endswith
__iendswith
__range
__date
__year
__month
__day
__weekday
__hour
__minute
__second
__isnull
__regex
__iregex
```

Aggregation Functions

- Calculate values over result set
- Use aggregate() plus calls to Count, Min, Max, etc
- Correspond to SQL COUNT(), MIN(), MAX(), etc

Some database tasks require calculations that use the entire result set (or subset). These are usually called aggregates. Common aggregation functions include count(), min(), and max().

To do this in Django, call the aggregate() function on a QuerySet, passing in one or more aggregation function. Each class is passed at least the name of the field to aggregate over.

aggregate() returns a dictionary where the keys are fieldname__aggregation_class, such as name__min. Parameters to aggregation functions may include a field name (positional), and the named parameter output_field, which specifies which field to return.

You can also generate aggregate values via the annotate() clause on a QuerySet

Aggregation Functions

```
Avg()
Count()
Min()
Max()
StdDev()
Sum()
Variance()
```


Example

```
Superhero.objects.aggregate(Count("name"))
{'name__count': 5}

Superhero.objects.aggregate(Min("name"))
{'name__min': 'Batman'}

Superhero.objects.aggregate(Max("name"))
{'name__max': 'Wonder Woman'}

Superhero.objects.aggregate(Min("name"),Max("name"))
{'name__max': 'Wonder Woman', 'name__min': 'Batman'}
```


Chaining filters

- Anything returning QuerySet can be chained
- Other methods are terminal (can't be chained)

Any QuerySet returned by some method can then have another method called on it; thus, you can chain filter methods to fine-tune what you need.

For instance, you could chain filter() and exclude(), then call count() on the result.

Example

```
Superhero.objects.filter(name__contains="man").count()
4

Superhero.objects.filter( name__contains="man").exclude(name__contains="woman")
[<Superhero: Superman>, <Superhero: Spiderman>, <Superhero: Batman>]

Superhero.objects.filter( name__contains="man").exclude( name__contains="woman").count()
3
```


Slicing QuerySets

- Slice operator [start:stop:step] works on QuerySets
- Slice is lazy – only retrieves data for slice
- Use to fetch first N objects, etc.
- Negative indices are not supported

While a QuerySet is not an actual list, it can be sliced like most builtin sequences. Furthermore, the slice returns another QuerySet, so it doesn't execute the actual query until the data is accessed.

One difference from normal slicing is that negative indices and ranges are not supported.

Example

```
Superhero.objects.all()[:3]
[<Superhero: Superman>, <Superhero: Spiderman>, <Superhero: Batman>]

Superhero.objects.filter(name__contains="man")[:2]
[<Superhero: Superman>, <Superhero: Spiderman>]
```

Related fields

- Search models via related fields
- Use column__foreign_column

To search models using values in related fields, use column__foreign_column__lookup. This lets you apply field lookups to fields in the related column.

Example

```
Superhero.objects.filter(enemies__name__icontains="Luthor").first().name
Superman
```


Q objects

- Encapsulates SQL expression in Python objects
- Only needed for complex queries

By default, chained queries are AND-ed together. If you need OR conditions, you can us the Q object. A Q object encapsulates (but does not execute) a SQL expression. Q objects can be combined with the | (OR) or & (AND) operators. Arguments to the Q object are the same as for filters – field lookups.

Example

superheroes/queries/views.py

```
q_hulk = Q(name__icontains="hulk")
q_woman = Q(name__icontains="woman")
...
Superhero.objects.filter(q_hulk | q_woman)
[<Superhero: Wonder Woman>, <Superhero: Hulk>]
```


Chapter 12 Exercises

Exercise 12-1 (dogs/*)

In this exercise, you will start a project that you will use throughout the rest of the class.

- Create a project named **dogs** using cookiecutter-rest
- In the dogs project, create an app named dogs_core
- Configure the app
- Create models for Dog and Breed
  - Dog fields
    - name
    - breed (foreign key)
    - sex (m or f)
    - is_neutered
  - Breed fields
    - name
- Update the database
  - Create migrations
  - Migrate
- Add models to admin

Now use the Django shell to add some records for dogs and breeds. Add at least 6 dogs and at least 2 breeds. Be sure to add lots of variations so you can use them for searching.

Once you have created the records and saved them, use the query methods to find records as follows (vary the queries to match your own data):

1. All dogs
2. All breeds
3. Dog with specified name
4. All female dogs
5. All dogs of a selected breed
6. All female dogs whose name begin with 'B' *etc*

NOTE

As an optional enhancement, you could add a Category model, with categories such as working, herding, companion, sporting, etc. See https://www.akc.org/public-education/resources/general-tips-information/dog-breeds-sorted-groups/ for a list of which dogs belong in which categories. Category would be a foreign key to Breed.

Appendix A: Django App Creation Checklist

- ▢ Create the project using cookiecutter (or other layout creator)

```
cookiecutter ../SETUP/cookiecutter-{django-version}
```

- ▢ Navigate to the project folder

```
cd project_name
```

- ▢ Run the first migration (optional for now)

```
python manage.py migrate
```

- ▢ Create the app using cookiecutter (or other layout creator)

```
cookiecutter ../SETUP/cookiecutter-{django-version}-app
```

- ▢ Add the app to INSTALLED_APPS in *project_name.project_name*.settings.py
- ▢ Create one or more models in *project_name.app_name*.models.py
- ▢ Create migrations and run them

```
python manage.py makemigrations
python manage.py migrate
```

- ▢ Register models in *project_name.project_name*.admin.py
- ▢ Create one or more views in *project_name.app_name*.views.py
- ▢ Create one or more templates in *project_name.app_name*.templates/*app_name*
- ▢ Add the views to the app's URL config in *project_name.app_name*.urls.py
- ▢ Add the app's URL config to the project's URL config in *project_name.project_name*.urls.py
- ▢ From the project folder, start the development server

```
python manage.py runserver
```

Index

O

P

Q

R

S

T

U

V

W

Z

