# Think Python (and other useful things)

Allen B. Downey (and Laurel Farris)

May 20, 2017

http://www.greenteapress.com/thinkpython/html/index.html

## Contents

# 1 The way of the program

The single most important skill for a computer scientist is **problem solving**: the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately.

. . .

# 2 Variables, expressions, and statements

## 2.1 Values and types

If you are not sure what type a value has, the interpreter can tell you:

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
>>> type(3.2)
<type 'float'>
```

## 2.2 Variables

An **assignment statement** creates new variables and gives them values.

## 2.3 Variable names and keywords

Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter.

The interpreter uses **keywords** to recognize the structure of the program, and they cannot be used as variable names.

Python 2 has 31 keywords:

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

## 2.4 Operators and operands

## 2.5 Expressions and statements

An **expression** is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable `x` has been assigned a value):

```
17
x
x + 17
```

A **statement** is a unit of code that the Python interpreter can execute. We have seen two kinds of statement: print and assignment.

Technically an expression is also a statement, but it is probably simpler to think of them as different things. The important difference is that an expression has a value; a statement does not.

## 2.6 Interactive mode and script mode

## 2.7 Order of operations

- Parentheses
- Exponents
- Multiplication/Division
- Addition/Subtraction

## 2.8 String operations

To perform a concatenation between two strings:

```
>> a = 1
>> print 'list_' + str(a) + '.txt'
list_1.txt
>> print 'Spam'*3
SpamSpamSpam
```

## 2.9 Comments

It is reasonable to assume that the reader can figure out *what* the code does; it is much more useful to explain *why*.

This comment is redundant with the code and useless:

```
v = 5     # assign 5 to v
```

This comment contains useful information that is not in the code:

```
v = 5     # velocity in meters/second.
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a tradeoff.

## 2.10 Debugging

# 3 Functions

## 3.1 Function calls

In the context of programming, a **function** is a named sequence of statements that performs a computation. To define a function, specify the name and sequence of statements. Then "call" the function by name later. Example of a **function call**:

```
>> type(32)
<type 'int'>
```

The name of the function is `type`. The expression in parentheses is called the **argument** of the function. The result, for this function, is the type of the argument.

It is common to say that a function "takes" an argument and "returns" a result. The result is called the **return value**.

## 3.2 Type conversion functions

Python provides built-in functions that convert values from one type to another. The `int` function takes any value and converts it to an integer:

```
>> int(‘32,)
32
>> int(3.999)
3
```

Note that decimals are truncated, not rounded.

The `float` function converts integers and strings to floating-point numbers:

```
>> float(32)
32.0
>> float(‘3.14159,)
3.14159
```

The `str` function converts its argument to a string:

```
>> str(32)
‘32,
>> str(3.14159)
‘3.14159,
```

## 3.3 Math functions

Python has a math module that provides most of the familiar mathematical functions. A **module** is a file that contains a collection of related functions. Modules must be imported before they can be used (such as `>>> import math`, which creates a **module object**) named math. If you print the module object, you get some information about it:

```
>> print math
<module ‘math, from ‘/usr/lib/python2.5/lib-dynload/math.so,>
```

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (this format is called **dot notation**).

```
>> ratio = signal_power / noise_power
>> decibels = 10 * math.log10(ratio)

>> radians = 0.7
>> height = math.sin(radians)
```

The first example computes the logarithm base 10 of the signal-to-noise ratio. The math module also provides a function called `log` that computes logarithms base `e`.

The second example finds the sine of `radians`. Trigonometric functions (`sin`, `cos`, `tan`, etc.) take arguments in radians.

The expression `math.pi` gets the variable `pi` from the math module, which is accurate to about 15 digits.

## 3.4 Composition

So far, we have looked at the elements of a programâĂŤvariables, expressions, and statementsâĂŤin isolation. One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, the argument of a function can be any kind of expression, including arithmetic operators and function calls:

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
x = math.exp(math.log(x+1))
```

## 3.5 Adding new functions

A **Function definition** specifies the name of a new function and the sequence of statements that execute when the function is called, e.g.:

```
def do_nothing():
    print "Doing nothing.,,
```

`def` is a keyword that indicates that this is a function definition. The name of the function is `do_nothing`. You should avoid having a variable and a function with the same name. The empty parentheses indicate that this function doesn't take any arguments.

The first line of the function definition is called the **header**; the rest is called the **body**. The header has to end with a colon and the body has to be indented. By convention, the indentation is always four spaces (see Section 3.13).

If you type a function definition in interactive mode, the interpreter prints ellipses (. . . ) to let you know that the definition isn't complete:

```
>> def print_lyrics():
... print "I,m a lumberjack, and I,m okay.,,
... print "I sleep all night and I work all day.,,
...
```

To end the function, enter an empty line (this is not necessary in a script).

Defining a function creates a variable with the same name.

```
>> print print_lyrics
<function print_lyrics at 0xb7e99e>
>> type(print_lyrics)
<type ‘function,>
```

The value of `print_lyrics` is a **function object**, which has type `‘function’`.

The syntax for calling the new function is the same as for built-in functions:

```
>> print_lyrics()
I,m a lumberjack, and I,m okay.
I sleep all night and I work all day.
```

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called `repeat_lyrics`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

And then call `repeat_lyrics`:

```
>> repeat_lyrics()
I,m a lumberjack, and I,m okay.
I sleep all night and I work all day.
I,m a lumberjack, and I,m okay.
I sleep all night and I work all day.
```

## 3.6 Definitions and uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```
def print_lyrics():
    print "I,m a lumberjack, and I,m okay.,,
    print "I sleep all night and I work all day.,,

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not get executed until the function is called, and the function definition generates no output. As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

## 3.7 Flow of execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the **flow of execution**.

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

WhatâĂŹs the moral of this sordid tale? When you read a program, you donâĂŹt always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

## 3.8 Parameters and arguments

Some of the built-in functions we have seen require arguments. For example, when you call `math.sin` you pass a number as an argument. Some functions take more than one argument: `math.pow` takes two, the base and the exponent.

Inside the function, the arguments are assigned to variables called **parameters**. Here is an example of a user-defined function that takes an argument:

```
def print_twice(bruce):
    print bruce
    print bruce
```

This function assigns the argument to a parameter named `bruce`. When the function is called, it prints the value of the parameter (whatever it is) twice.

This function works with any value that can be printed.

```
>> print_twice(,Spam,)
Spam
Spam
>> print_twice(17)
```

```
17
17
>> print_twice(math.pi)
3.14159265359
3.14159265359
```

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for `print_twice`:

```
>> print_twice(,Spam ,*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in the examples the expressions `'Spam '*4` and `math.cos(math.pi)` are only evaluated once.

You can also use a variable as an argument:

```
>> michael = ,Eric, the half a bee.,
>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

The name of the variable we pass as an argument (`michael`) has nothing to do with the name of the parameter (`bruce`). It doesnâĂŹt matter what the value was called back home (in the caller); here in `print_twice`, we call everybody `bruce`.

## 3.9 Variables and parameters are local

When you create a variable inside a function, it is **local**, which means that it only exists inside the function. For example:

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

```
>> line1 = ,Bing tiddle ,
>> line2 = ,tiddle bang.,
>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

When `cat_twice` terminates, the variable `cat` is destroyed. If we try to print it, we get an exception:

```
>> print cat
NameError: name ,cat, is not defined
```

Parameters are also local. For example, outside `print_twice`, there is no such thing as `bruce`.

## 3.10 Stack diagrams

Meh.

## 3.11 Fruitful functions and void functions

Some of the functions we are using, such as the math functions, yield results; for lack of a better name, I call them **fruitful functions**. Other functions, like `print_twice`, perform an action but donâĂŹt return a value. They are called **void functions**.

When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

When you call a function in interactive mode, Python displays the result:

```
>> math.sqrt(5)
2.2360679774997898
```

But in a script, if you call a fruitful function all by itself, the return value is lost forever.

```
math.sqrt(5)
```

This script computes the square root of 5, but since it doesnâĂŹt store or display the result, it is not very useful.

Void functions might display something on the screen or have some other effect, but they donâĂŹt have a return value. If you try to assign the result to a variable, you get a special value called `None`.

```
>> result = print_twice(,Bing,)
Bing
Bing
>> print result
None
```

The value `None` is not the same as the string `'None'`. It is a special value that has its own type:

```
>> print type(None)
<type ,NoneType,>
```

The functions we have written so far are all void. We will start writing fruitful functions in a few chapters.

## 3.12 Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

## 3.13 Importing with `from`

Python provides two ways to import modules; we have already seen one:

```
>> import math
>> print math
<module ,math, (built-in)>
>> print math.pi
3.14159265359
```

If you import `math`, you get a module object named `math`. The module object contains constants like `pi` and functions like `sin` and `exp`.

But if you try to access `pi` directly, you get an error.

```
>> print pi
Traceback (most recent call last):
  File "<stdin>,,, line 1, in <module>
NameError: name ‘pi, is not defined
```

As an alternative, you can import an object from a module like this:

```
>> from math import pi
```

Now you can access pi directly, without dot notation.

```
>> print pi
3.14159265359
```

Or you can use the star operator to import *everything* from the module:

```
>> from math import *
>> cos(pi)
-1.0
```

The advantage of importing everything from the math module is that your code can be more concise. The disadvantage is that there might be conflicts between names defined in different modules, or between a name from a module and one of your variables.

## 3.14 Debugging

If you are using a text editor to write your scripts, you might run into problems with spaces and tabs. The best way to avoid these problems is to use spaces exclusively (no tabs). Most text editors that know about Python do this by default, but some donâĂŹt.

Tabs and spaces are usually invisible, which makes them hard to debug, so try to find an editor that manages indentation for you.

Also, donâĂŹt forget to save your program before you run it. Some development environments do this automatically, but some donâĂŹt. In that case the program you are looking at in the text editor is not the same as the program you are running.

Debugging can take a long time if you keep running the same, incorrect, program over and over!

Make sure that the code you are looking at is the code you are running. If youâĂŹre not sure, put something like `print 'hello'` at the beginning of the program and run it again. If you don't see `hello`, you're not running the right program!

# 4 Case study: interface design

## 4.1 TurtleWorld

A **package** is a collection of modules.

```
from package_name.module_name import *
```

This should import all functions from the module. If you downloaded the Package modules but did not install them as a package, you can either work in the directory that contains the Package files, or add that directory to PythonâĂŹs search path. Then you can import Module like this:

```
from Module import *


from Package.Module import *
var1 = Module()
var2 = Instance()  # One of several possible instances
func(var2, args)
```

### 4.2 Simple repetition

### 4.3 Exercises

### 4.4 Encapsulation

### 4.5 Generalization

### 4.6 Interface design

### 4.7 Refactoring

### 4.8 A development plan

### 4.9 docstring

### 4.10 Debugging

## 5 Conditionals and recursion

### 5.1 Modulus operator

The **Modulus operator** works on integers and yields the remainder when the first operand is divided by the second.

```
>>> quotient = 7 / 3
>>> print quotient
2
>>> remainder = 7 % 3
>>> print remainder
1
```

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by anotherâĂŤif x % y is zero, then x is divisible by y.

Also, you can extract the right-most digit or digits from a number. For example, x % 10 yields the right-most digit of x (in base 10). Similarly x % 100 yields the last two digits.

### 5.2 Boolean expressions

```
>>> 5 == 5
True
>>> 5 == 6
False
```

True and False are special values that belong to the type `bool`; they are not strings.

```
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
```

**Relational operators:**

- ==
- !=
- >
- <
- >=
- <=

## 5.3 Logical operators

- `and`
- `or`
- `not`

The `not` operator negates a boolean expression, so `not (x > y)` is true if `x > y` is false, that is, if `x` is less than or equal to `y`.

Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as "true."

```
>>> 17 and True
True
```

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it (unless you know what you are doing).

## 5.4 Conditional execution

The `if` **statment:**

```
if x > 0:
    print 'x is positive'
```

The boolean expression after `if` is called the **condition**. If it is true, then the indented statement gets executed. If not, nothing happens. `if` statements have the same structure as function definitions: a header followed by an indented body. Statements like this are called **compound statements**.

There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet). In that case, you can use the `pass` statement, which does nothing.

```
if x < 0:
    pass    # Need to handle negative values
```

## 5.5 Alternative execution

```
if x%2 == 0:
    print 'x is even'
else:
    print 'x is odd'
```

Since the condition must be true or false, exactly one of the alternatives will be executed. The alternatives are called **branches**, because they are branches in the flow of execution.

## 5.6 Chained conditionals

```
if x < y:
    print 'x is less than y'
elif x > y
    print 'x is less than y'
else:
    print 'x and y are equal'
```

If there is an `else` clause, it has to be at the end, but there doesn't have to be one.

## 5.7 Nested conditionals

Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. In general, it is a good idea to avoid them when you can. *Logical operators* often provide a way to simplify nested conditional statements.

## 5.8 Recursion

**Recursive** functions call themselves:

```
def countdown(n):
    if n <= 0:
        print 'Blastoff!'
    else:
        print n
        countdown(n-1)
```

Sometimes it's easier to write a recursive function than to use `for` loops.

## 5.9 Stack diagrams

## 5.10 Infinite recursion

## 5.11 Keyboard input

Python 2 provides a built-in function called `raw_input` that gets input from the keyboard. In Python 3, it is called `input`. When this function is called, the program stops and waits for the user to type something. When the user presses Return or Enter, the program resumes and `raw_input` returns what the user typed as a string.

```
>>> text = raw_input()
Blah
>>> print text
Blah

>>> name = raw_input('What... is your name?\n')
What... is your name?
Arthur, King of the Britons!
>>> print name
Arthur, King of the Britons!
```

## 5.12 Debugging

# 6 Fruitful functions

# 7 Iteration

## 7.1 Multiple assignment

## 7.2 Updating variables

## 7.3 The `while` statement

```
def countdown(n):
    while n > 0:
        print n
        n = n-1
    print 'Blastoff!'
```

## 7.4 `break`

Suppose you want to take input from the user until they type `done`. You could write:

```
while True:
    line = raw_input('> ')
    if line == 'done':
        break
    print line

print 'Done!'
```

This way of writing while loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens.").

## 7.5 Square roots

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.

For example, one way of computing square roots is NewtonâĂŹs method. Suppose that you want to know the square root of $a$. If you start with almost any estimate, $x$, you can compute a better estimate with the following formula:

$$y = \frac{x + a/x}{2}$$

For example, if $a$ is 4 and $x$ is 3:

```
>>> a = 4.0
>>> x = 3.0
>>> y = (x + a/x) / 2
>>> print y
2.16666666667
```

Which is closer to the correct answer ($\sqrt{4} = 2$). If we repeat the process with the new estimate, it gets even closer. In general we donâĂŹt know ahead of time how many steps it takes to get to the right answer, but we know when we get there because the estimate stops changing (like my alignment codes!).

For most values of a this works fine, but in general it is dangerous to test `float` equality. Floating-point values are only approximately right: most rational numbers, like 1/3, and irrational numbers, like $\sqrt{2}$, canâĂŹt be represented exactly with a `float`.

Rather than checking whether $x$ and $y$ are exactly equal, it is safer to use the built-in function `abs` to compute the absolute value, or magnitude, of the difference between them:

```
if abs(y-x) < epsilon:
    break
```

where `epsilon` has a value like 0.0001 that determines how close is close enough.

## 7.6 Algorithms

NewtonâĂŹs method is an example of an **algorithm**: it is a mechanical process for solving a category of problems (in this case, computing square roots).

It is not easy to define an algorithm. It might help to start with something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic.

But if you were "lazy," you probably cheated by learning a few tricks. For example, to find the product of n and 9, you can write nâĹŠ1 as the first digit and 10âĹŠn as the second digit. This trick is a general solution for multiplying any single-digit number by 9. ThatâĂŹs an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules.

In my opinion, it is embarrassing that humans spend so much time in school learning to execute algorithms that, quite literally, require no intelligence.

On the other hand, the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming.

Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of an algorithm.

## 7.7 Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more place for bugs to hide.

One way to cut your debugging time is âĂIJdebugging by bisection.âĂİ For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a print statement (or something else that has a verifiable effect) and run the program.

If the mid-point check is incorrect, there must be a problem in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is fewer than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the âĂIJmiddle of the programâĂİ is and not always possible to check it. It doesnâĂŹt make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

# 8 Strings

## 8.1 A string is a sequence

## 8.2 `len`

## 8.3 Traversal with a `for` loop

## 8.4 String slices

## 8.5 Strings are immutable

## 8.6 Searching

## 8.7 Looping and counting

## 8.8 String methods

A **method** is similar to a function - it takes arguments and returns a value, but the syntax is different. For example, the method `upper` takes a string and returns a new string with all uppercase letters:

Instead of the function syntax `upper(word)`, it uses the method syntax `word.upper()`.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print new_word
BANANA
```

This form of dot notation specifies the name of the method, `upper`, and the name of the string to apply the method to, `word`. The empty parentheses indicate that this method takes no argument.

A method call is called an **invocation**; in this case, we would say that we are invoking `upper` on the `word`.

As it turns out, there is a string method named `find` that is remarkably similar to the function we wrote:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print index
1
```

In this example, we invoke `find` on `word` and pass the letter we are looking for as a parameter.

Actually, the `find` method is more general than our function; it can find substrings, not just characters:

```
>>> word.find('na')
2
```

It can take as a second argument the index where it should start:

```
>>> word.find('na', 3)
4
```

And as a third argument the index where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

This search fails because `b` does not appear in the index range from `1` to `2` (not including `2`).

### 8.9 The `in` operator

### 8.10 String comparison

### 8.11 Debugging

# 9 Case study: word play

# 10 Lists

## 10.1 A list is a sequence

Like a string, a **list** is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called **elements** or sometimes **items**.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets:

```
[10, 20, 30, 40]
[‚crunchy frog‚, ‚ram bladder‚, ‚lark vomit‚]
```

*The elements of a list don't have to be the same type*. The following list contains a string, a float, an integer, and (lo!) another list:

```
[‚spam‚, 2.0, 5, [10, 20]]
```

A list within another list is **nested**.

A list that contains no elements is called an *empty list*; you can create one with empty brackets, [].

As you might expect, you can assign list values to variables:

```
>> cheeses = [‚Cheddar‚, ‚Edam‚, ‚Gouda‚]
>> numbers = [17, 123]
>> empty = []
>> print cheeses, numbers, empty
[‚Cheddar‚, ‚Edam‚, ‚Gouda‚] [17, 123] []
```

## 10.2 Lists are mutable

The syntax for *accessing* the elements of a list is the same as for accessing the characters of a string - the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>> print cheeses[0]
Cheddar
```

*Unlike strings*, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>> numbers = [17, 123]
>> numbers[1] = 5
>> print numbers
[17, 5]
```

The one-eth element of numbers, which used to be 123, is now 5.

You can think of a list as a relationship between indices and elements. This relationship is called a **mapping**; each index "maps to" one of the elements.

List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an `IndexError`.
- If an index has a negative value, it counts backward from the end of the list.

The `in` operator also works on lists.

```
>> cheeses = [‘Cheddar‚, ‘Edam‚, ‘Gouda‚]
>> ‘Edam‚ in cheeses
True
>> ‘Brie‚ in cheeses
False
```

## 10.3 Traversing a list

The most common way to traverse the elements of a list is with a `for` loop. The syntax is the same as for strings:

```
for cheese in cheeses:
    print cheese
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions `range` and `len`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element. `len` returns the number of elements in the list. `range` returns a list of indices from 0 to $n-1$, where $n$ is the length of the list. Each time through the loop `i` gets the index of the next element. The assignment statement in the body uses `i` to read the old value of the element and to assign the new value.

A for `loop` over an *empty* list never executes the body:

```
for x in []:
    print ‘This never happens.,
```

Although a list can contain another list, *the nested list still counts as a single element*. The length of this list is 4:

```
[‘spam,, 1, [‘Brie,, ‘Roquefort,, ‘Pol le Veq,], [1, 2, 3]]
```

## 10.4 List operations

IMPORTANT: these are not mathematical operations for lists. Convert to numpy arrays: `a = np.array(a)`

The following doesn't work because x is a list, not an array.

```
>> x = [1,2,3]
>> y = x**2
```

The + operator concatenates lists:

```
>> a = [1, 2, 3]
>> b = [4, 5, 6]
>> c = a + b
>> print c
[1, 2, 3, 4, 5, 6]
```

Similarly, the $*$ operator repeats a list a given number of times:

```
>> [0] * 4
[0, 0, 0, 0]
>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats `[0]` four times. The second example repeats the list `[1,2,3]` three times.

## 10.5 List slices

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]  # Print indices 1 and 2
['b', 'c']
>>> t[:4]  # Print indices 0 - 3
```

```
['a', 'b', 'c', 'd']
>>> t[3:]  # Print indices 3 - end
['d', 'e', 'f']
>>> t[:]  # Print entire list
['a', 'b', 'c', 'd', 'e', 'f']

>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']  # Update multiple elements!
>>> print t
['a', 'x', 'y', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle or mutilate lists.

## 10.6 List methods

```
list1.append('d')    # append element 'd' to the end of list1
list1.extend(list2)  # append elements of list2 to end of list1
list1.sort()         # arrange elments from low to high
```

Appending a list to another list (rather than using extend) will append a 'sublist' instead of simply appending the individual elements to the end.

List methods are all void; they modify the list and return `None`. If you accidentally write `t = t.sort()`, you will be disappointed with the result.

## 10.7 Map, filter, and reduce

```
>>> t = [1,2,3]
>>> sum(t)
6
```

Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

`res` is initialized with an empty list; each time through the loop, the next element is appended. So `res` is another kind of accumulator.

An operation like `capitalize_all` is sometimes called a **map** because it "maps" a function (in this case, the method `capitalize`) onto each of the elements in a sequence.

Another common operation is to select some of the elements from a list and return a sublist. For example, the following function takes a list of strings and returns a list that contains only the uppercase strings:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

`isupper` is a string method that returns `True` if the string contains only uppercase letters. An operation like `only_upper` is called a **filter** because it selects some of the elements and filters out the others.

Most common list operations can be expressed as a combination of map, filter and reduce. Because these operations are so common, Python provides language features to support them, including the built-in function `map` and an operator called a "list comprehension."

## 10.8 Deleting elements

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print t
```

```
['a', 'c']
>>> print x
b
```

`pop` modifies the list and returns the element that was removed. If you donâĂŹt provide an index, it deletes and returns the last element.

If you donâĂŹt need the removed value, you can use the `del` operator:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print t
['a', 'c']
```

The return value from `remove` is `None`.

## 10.9 Lists and strings

## 10.10 Objects and values

## 10.11 Aliasing

## 10.12 List arguments

## 10.13 Debugging

# 11 Dictionaries

A **dictionary** is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.

# 12 Tuples

## 12.1 Tuples are immutable

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. *The important difference is that tuples are immutable*.

Syntactically, a tuple is a comma-separated list of values:

```
>> t = (‘a,, ‘b,,‘c,,‘d,,‘e,)
```

Parentheses are common, but not necessary.

To create a tuple with a single element, you have to include a final comma:

```
>> t1 = ‘a,,
>> type(t1)
<type ‘tuple,>
```

A value in parentheses is not a tuple:

```
>> t2 = (‘a,)
>> type(t2)
<type ‘str,>
```

Another way to create a tuple is the built-in function `tuple`. With no argument, it creates an empty tuple:

```
>> t = tuple()
>> print t
()
```

If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

```
>> t = tuple(,lupins,)
>> print t
(,l,, ,u,, ,p,, ,i,, ,n,, ,s,)
```

Most list operators also work on tuples. The bracket operator indexes an element:

```
>> t = (,a,, ,b,, ,c,, ,d,, ,e,)
>> print t[0]
,a,
```

And the slice operator selects a range of elements.

```
>> print t[1:3]
(,b,, ,c,)
```

But if you try to modify one of the elements of the tuple, you get an error:

```
>> t[0] = ‘A,
TypeError: object doesn,t support item assignment
```

You canâĂŹt modify the elements of a tuple, but you can *replace* one tuple with another:

```
>> t = ('A',) + t[1:]
>> print t
('A', 'b', 'c', 'd', 'e')
```

## 12.2 Tuple assignment

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap `a` and `b`:

```
>> temp = a
>> a = b
>> b = temp
```

This solution is cumbersome; **tuple assignment** is more elegant:

```
>> a, b = b, a
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right have to be the same:

```
>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, you could write:

```
>> addr = 'monty@python.org'
>> uname, domain = addr.split('\@')
```

The return value from `split` is a list with two elements; the first element is assigned to `uname`, the second to `domain`.

```
>> print uname
monty
>> print domain
python.org
```

## 12.3 Tuples as return values

Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute `x/y` and then `x%y`. It is better to compute them both at the same time.

The built-in function `divmod` takes two arguments and returns a tuple of two values, the quotient and remainder. You can store the result as a tuple:

```
>> t = divmod(7, 3)
>> print t
(2, 1)
```

Or use tuple assignment to store the elements separately:

```
>> quot, rem = divmod(7, 3)
>> print quot
2
>> print rem
1
```

Here is an example of a function that returns a tuple:

```
def min_max(t):
    return min(t), max(t)
```

max and min are built-in functions that find the largest and smallest elements of a sequence. `min_max` computes both and returns a tuple of two values.

## 12.4 Variable-length argument tuples

Functions can take a variable number of arguments. A parameter name that begins with * **gathers** arguments into a tuple. For example, `printall` takes any number of arguments and prints them:

```
def printall(*args):
    print args
```

The gather parameter can have any name you like, but `args` is conventional. HereâĂŹs how the function works:

```
>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

The complement of gather is `scatter`. If you have a sequence of values and you want to pass it to a function as multiple arguments, you can use the * operator. For example, `divmod` takes exactly two arguments; it doesnâĂŹt work with a tuple:

```
>> t = (7, 3)
>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

But if you scatter the tuple, it works:

```
>> divmod(*t)
(2, 1)
```

## 12.5 Lists and tuples

`zip` is a built-in function that takes two or more sequences and "zips" them into a list of tuples where each tuple contains one element from each sequence. In Python 3, `zip` returns an iterator of tuples, but for most purposes, an iterator behaves like a list.

This example zips a string and a list:

```
>> s = 'abc'
>> t = [0, 1, 2]
>> zip(s, t)
[('a', 0), ('b', 1), ('c', 2)]
```

The result is a list of tuples where each tuple contains a character from the string and the corresponding element from the list.

If the sequences are not the same length, the result has the length of the shorter one.

```
>> zip('Anne', 'Elk')
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

You can use tuple assignment in a `for` loop to traverse a list of tuples:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print number, letter
```

Each time through the loop, Python selects the next tuple in the list and assigns the elements to `letter` and `number`. The output of this loop is:

```
0 a
1 b
2 c
```

If you combine `zip`, for and tuple assignment, you get a useful idiom for traversing two (or more) sequences at the same time. For example, `has_match` takes two sequences, `t1` and `t2`, and returns `True` if there is an index `i` such that `t1[i] == t2[i]`:

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

If you need to traverse the elements of a sequence and their indices, you can use the built-in function `enumerate`:

```
for index, element in enumerate('abc'):
    print index, element
```

The output of this loop is:

```
0 a
1 b
2 c
```

Again.

## 12.6 Dictionaries and tuples

Dictionaries have a method called `items` that returns a list of tuples, where each tuple is a key-value pair.

```
>> d = {'a':0, 'b':1, 'c':2}
>> t = d.items()
>> print t
[('a', 0), ('c', 2), ('b', 1)]
```

As you should expect from a dictionary, the items are in no particular order. In Python 3, `items` returns an iterator, but for many purposes, iterators behave like lists.

Going in the other direction, you can use a list of tuples to initialize a new dictionary:

```
>> t = [('a', 0), ('c', 2), ('b', 1)]
>> d = dict(t)
>> print d
{'a': 0, 'c': 2, 'b': 1}
```

Combining `dict` with `zip` yields a concise way to create a dictionary:

```
>> d = dict(zip('abc', range(3)))
>> print d
{'a': 0, 'c': 2, 'b': 1}
```

The dictionary method `update` also takes a list of tuples and adds them, as key-value pairs, to an existing dictionary.

Combining `items`, tuple assignment and `for`, you get the idiom for traversing the keys and values of a dictionary:

```
for key, val in d.items():
    print val, key
```

The output of this loop is:

```
0 a
2 c
1 b
```

Again. It is common to use tuples as keys in dictionaries (primarily because you canâĂŹt use lists). For example, a telephone directory might map from last-name, first-name pairs to telephone numbers. Assuming that we have defined `last`, `first` and `number`, we could write:

```
directory[last,first] = number
```

The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

```
for last, first in directory:
    print first, last, directory[last,first]
```

This loop traverses the keys in `directory`, which are tuples. It assigns the elements of each tuple to `last` and `first`, then prints the name and corresponding telephone number.

## 12.7 Comparing tuples

The relational operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
>> (0, 1, 2) < (0, 3, 4)
True
>> (0, 1, 2000000) < (0, 3, 4)
True
```

The `sort` function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on.

This feature lends itself to a pattern called **DSU** for

| | |
|---|---|
| `Decorate` | a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence, |
| `Sort` | the list of tuples, and |
| `Undecorate` | by extracting the sorted elements of the sequence. |

For example, suppose you have a list of words and you want to sort them from longest to shortest:

```
def sort_by_length(words):
    t = []
    for word in words:
        t.append((len(word), word))

    t.sort(reverse=True)

    res = []
    for length, word in t:
        res.append(word)
    return res
```

The first loop builds a list of tuples, where each tuple is a word preceded by its length.

`sort` compares the first element, length, first, and only considers the second element to break ties. The keyword argument `reverse=True` tells `sort` to go in decreasing order.

The second loop traverses the list of tuples and builds a list of words in descending order of length.

## 12.8 Sequences of sequences

I have focused on lists of tuples, but almost all of the examples in this chapter also work with lists of lists, tuples of tuples, and tuples of lists. To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences.

In many contexts, the different kinds of sequences (strings, lists and tuples) can be used interchangeably. So how and why do you choose one over the others?

To start with the obvious, strings are more limited than other sequences because the elements have to be characters. They are also immutable. If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

1. In some contexts, like a `return` statement, it is syntactically simpler to create a tuple than a list. In other contexts, you might prefer a list.
2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.
3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

Because tuples are immutable, they donâĂŹt provide methods like `sort` and `reverse`, which modify existing lists. But Python provides the built-in functions `sorted` and `reversed`, which take any sequence as a parameter and return a new list with the same elements in a different order.

## 12.9 Debugging

Lists, dictionaries and tuples are known generically as **data structures**; in this chapter we are starting to see compound data structures, like lists of tuples, and dictionaries that contain tuples as keys and lists as values. Compound data structures are useful, but they are prone to what I call **shape errors**; that is, errors caused when a data structure has the wrong type, size or composition. For example, if you are expecting a list with one integer and I give you a plain old integer (not in a list), it wonâĂŹt work.

To help debug these kinds of errors, I have written a module called `structshape` that provides a function, also called `structshape`, that takes any kind of data structure as an argument and returns a string that summarizes its shape. You can download it from http://thinkpython.com/code/structshape.py

HereâĂŹs the result for a simple list:

```
>> from structshape import structshape
>> t = [1,2,3]
>> print structshape(t)
list of 3 int
```

A fancier program might write “list of 3 ints,” but it was easier not to deal with plurals. HereâĂŹs a list of lists:

```
>> t2 = [[1,2], [3,4], [5,6]]
>> print structshape(t2)
list of 3 list of 2 int
```

If the elements of the list are not the same type, `structshape` groups them, in order, by type:

```
>> t3 = [1, 2, 3, 4.0, ,5,, ,6,, [7], [8], 9]
>> print structshape(t3)
list of (3 int, float, 2 str, 2 list of int, int)
```

Here's a list of tuples:

```
>> s = ,abc,
>> lt = zip(t, s)
>> print structshape(lt)
list of 3 tuple of (int, str)
```

And here's a dictionary with 3 items that map integers to strings.

```
>> d = dict(lt)
>> print structshape(d)
dict of 3 int->str
```

If you are having trouble keeping track of your data structures, `structshape` can help.

# 13 Case study: data structure selection

# 14 Files

http://www.greenteapress.com/thinkpython/html/thinkpython015.html

## 14.1 Persistence

Most of the programs we have seen so far are transient in the sense that they run for a short time and produce some output, but when they end, their data disappears. If you run the program again, it starts with a clean slate.

Other programs are **persistent**: they run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off.

Examples of persistent programs are operating systems, which run pretty much whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network.

One of the simplest ways for programs to maintain their data is by reading and writing text files. We have already seen programs that read text files; in this chapter we will see programs that write them.

An alternative is to store the state of the program in a database. In this chapter I will present a simple database and a module, `pickle`, that makes it easy to store program data.

## 14.2 Reading and writing

```
file:
f=open('file.txt','[rwa]') [read,write,append]
s=f.readline()
f.close(),
alternatively:
for line in f:
  print line
```

The `write` method puts data into the file. To write a file, you have to open it with mode 'w' as a second parameter:

```
>>> fout = open('output.txt', 'w')
>>> print fout
    <open file 'output.txt', mode 'w' at 0
    x7fa44614c420>
>>> line1 = ''This here's the wattle,\n''
>>> fout.write(line1)
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created. The file object keeps track of where it is, so if you call `write` again, it adds the new data to the end.

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
```

When you are done writing, you have to close the file.

```
>>> fout.close()
```

## 14.3 Format operator

```
'''
Formatted output
 > print 'the number is {:.#e|:nd|:n.nf|:ns}'.
    format(x)
    exponential, integer, float, string
 General syntax:
'''
template.format(var_1,var_2,...var_n)
# template:
#'{[field][!conversion]:[spec]}'
field = index of variables listed in .format()
```

```
conversion = int, float, string, etc.
spec = specifier
[[fill]align][sign][#][0][minwidth][.prec][type
    ]
align:
<(left,default)
>(right)
=(padding after sign, before digits)
^(center)
0: zero padding (same as '=' and fill char of
    0)
type: e,f,g(general)

# Including text:
print 'Variable 2 is {1} and variable 1 is {0}'
    .format(var_1,var_2)

'{:8d}asdfad{:8.2f}'.format(i,f)
formats: {:nd},{:n.nf},{:ns}
#    n-number (width.precision), d-integer, f-
    float, s-string
#---------- How to use a variable for width
    ?? ----------#
```

The argument of `write` has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with `str`:

```
>>> x = 52
>>> fout.write(str(x))
```

An alternative is to use the **format operator**, `%`. When applied to integers, `%` is the modulus operator. But when the first operand is a string, `%` is the format operator.

The first operand is the **format string**, which contains one or more **format sequences**, which specify how the second operand is formatted. The result is a string.

For example, the format sequence `'%d'` means that the second operand should be formatted as an integer (`d` stands for "decimal"):

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string `'42'`, which is not to be confused with the integer value 42.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses `'%d'` to format an integer, `'%g'` to format a floating-point number, and `'%s'` to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3,
    0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format
    string
>>> '%d' % 'dollars'
TypeError: illegal argument type for built-in
    operation
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

The format operator is powerful, but it can be difficult to use. You can read more about it at http://docs.python.org/2/library/stdtypes.html#string-formatting.

## 14.4 Filenames and paths

The `os` ('operating system') module provides functions for working with files and directories. `os.getcwd` returns the current working directory.

```
>>> import os
>>> cwd = os.getcwd()
>>> print cwd
/home/dinsdale
```

To find the absolute path to a *file*:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

`os.path.exists` checks whether a file or directory exists.

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

If it exists, `os.path.isdir` checks whether it's a directory:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('music')
True
```

Similarly, `os.path.isfile` checks whether it's a file:

`os.listdir` returns a list of the files (and other directories) in the given directory:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

To demonstrate these functions, the following example "walks" through a directory, prints the names of all the files, and calls itself recursively on all the directories.

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print path
        else: walk(path)
```

`os.path.join` takes a directory and a file name and joins them into a complete path.

## 14.5 Catching exceptions

The `try` statement:

```
try:
    fin = open('bad_file')
    for line in fin:
        print line
    fin.close()
except:
    print 'Something went wrong.'
```

The syntax is similar to an `if` statement. Python starts by executing the try clause. If all goes well, it skips the `except` clause and proceeds. If an exception occurs, it jumps out of the `try` clause and executes the except clause.

Handling an exception with a try statement is called **catching** an exception. In this example, the `except` clause prints an error message that is not very helpful. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

## 14.6 Databases

A **database** is a file that is organized for storing data. Most databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference is that the database is on disk (or other permanent storage), so it persists after the program ends.

The module `anydbm` provides an interface for creating and updating database files. As an example, I'll create a database that contains captions for image files.

Opening a database is similar to opening other files:

```
>>> import anydbm
>>> db = anydbm.open('captions.db', 'c')
```

The mode `'c'` means that the database should be created if it doesn't already exist. The result is a database object that can be used (for most operations) like a dictionary. If you create a new item, `anydbm` updates the database file.

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

When you access one of the items, `anydbm` reads the file:

```
>>> print db['cleese.png']
Photo of John Cleese.
```

If you make another assignment to an existing key, `anydbm` replaces the old value:

```
>>> db['cleese.png'] = 'Photo of John Cleese
    doing a silly walk.'
>>> print db['cleese.png']
Photo of John Cleese doing a silly walk.
```

Many dictionary methods, like `keys` and `items`, also work with database objects. So does iteration with a `for` statement.

```
for key in db:
    print key
```

As with other files, you should close the database when you are done:

```
>>> db.close()
```

## 14.7 Pickling

A limitation of `anydbm` is that the keys and values have to be strings. The `pickle` module translates almost any type of object into a string suitable for storage in a database, and then translates strings back into objects. `pickle.dumps` takes an object as a parameter and returns a string representation (`dumps` is short for "dump string"):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
'(lp0\nI1\naI2\naI3\na.'
```

`pickle.loads` reconstitutes the object:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> print t2
[1, 2, 3]
```

Although the new object has the same value as the old, it is not (in general) the same object:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

In other words, pickling and then unpickling has the same effect as copying the object.

You can use `pickle` to store non-strings in a database. In fact, this combination is so common that it has been encapsulated in a module called `shelve`.

From my notes:

```
pickle.dump(var_name, open(''save.p'', ''wb''))
var\_name = pickle.load(open(''save.p'', ''rb''
    ))
```

## 14.8 Pipes

Any program that you can launch from the shell can also be launched from Python using a **pipe**. A pipe is an object that represents a running program.

You can launch `ls -l` with `popen`:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

The argument is a string that contains a shell command. The return value is an object that behaves just like an open file. You can read the output from the `ls` process one line at a time with `readline` or get the whole thing at once with `read`:

```
>>> res = fp.read()
```

When you are done, you close the pipe like a file:

```
>>> stat = fp.close()
>>> print stat
None
```

The return value is the final status of the `ls` process; None means that it ended normally (with no errors).

For example, most Unix systems provide a command called `md5sum` that reads the contents of a file and computes a "checksum". You can read about MD5 at http://en.wikipedia.org/wiki/Md5. This command provides an efficient way to check whether two files have the same contents. The probability that different contents yield the same checksum is very small (that is, unlikely to happen before the universe collapses).

You can use a pipe to run `md5sum` from Python and get the result:

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print res
1e0033f0ed0656636de0d75144ba32e0  book.tex
>>> print stat
None
```

## 14.9 Writing modules

Any file that contains Python code can be imported as a module. For example, suppose you have a file named `wc.py` with the following code:

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print linecount('wc.py')
```

You can import this program:

```
>>> import wc
7
```

Now you have a module object `wc` that provides a function called `linecount`:

```
>>> print wc
<module 'wc' from 'wc.py'>
>>> wc.linecount('wc.py')
7
```

The only problem with this example is that when you import the module it executes the test code at the bottom. Normally when you import a module, it defines new functions but it doesn't execute them.

Programs that will be imported as modules often use the following idiom:

```
if __name__ == '__main__':
    print linecount('wc.py')
```

`__name__` is a built-in variable that is set when the program starts. If the program is running as a script, `__name__` has the value `__main__`; in that case, the test code is executed. Otherwise, if the module is being imported, the test code is skipped.

## 14.10 Debugging

When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because spaces, tabs, and newlines are normally invisible:

```
>>> s = '1 2\t 3\n 4'
>>> print s
1 2  3
 4
```

The built-in function `repr` takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

```
>>> print repr(s)
'1 2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented `\n`. Others use a return character, represented `\r`. Some use both. If you move files between different systems, these inconsistencies might cause problems.

For most systems, there are applications to convert from one format to another. You can find them (and read more about this issue) at http://en.wikipedia.org/wiki/Newline. Or, of course, you could write one yourself.

# 15 Classes and objects

## 15.1 User-defined types

As an example, we will create a type called `Point` that represents a point in two-dimensional space. In mathematical notation, points are often written as $(x, y)$. There are several ways we might represent points in Python:

- We could store the coordinates separately in two variables, x and y.
- We could store the coordinates as elements in a list or tuple.
- We could create a new type to represent points as objects.

A **class** is a user-defined type. A class definition looks like this:

```
class Point(object):
    """Represents a point in 2-D space."""
```

This header indicates that the new class is a `Point`, which is a kind of `object`, which is a built-in type.

The body is a docstring that explains what the class is for. You can define variables and functions inside a class definition,

Defining a class named `Point` creates a class object.

```
>>> print Point
<class '__main__.Point'>
```

Because `Point` is defined at the top level, its "full name" is `__main__.Point`.

The class object is like a factory for creating objects. To create a Point, you call `Point` as if it were a function.

```
>>> blank = Point()
>>> print blank
<__main__.Point instance at 0xb7e9d3ac>
```

The return value is a reference to a Point object, which we assign to `blank`. Creating a new object is called **instantiation**, and the object is an **instance** of the class.

When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix `0x` means that the following number is in hexadecimal).

## 15.2 Attributes

You can assign values to an instance using dot notation:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

This syntax is similar to the syntax for selecting a variable from a module, such as `math.pi` or `string.whitespace`. In this case, though, we are assigning values to named elements of an object. These elements are called **attributes**.

The variable `blank` refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number.

You can read the value of an attribute using the same syntax:

```
>>> print blank.y
4.0
>>> x = blank.x
>>> print x
3.0
```

The expression `blank.x` means, "Go to the object `blank` refers to and get the value of `x`." In this case, we assign that value to a variable named `x`. There is no conflict between the variable `x` and the attribute `x`.

You can use dot notation as part of any expression. For example:

```
>> print '(%g, %g)' % (blank.x, blank.y)
(3.0, 4.0)
>> distance = math.sqrt(blank.x**2 + blank.y**2)
>> print distance
5.0
```

You can pass an instance as an argument in the usual way. For example:

```
def print_point(p):
    print '(%g, %g)' % (p.x, p.y)
```

`print_point` takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass `blank` as an argument:

```
>>> print_point(blank)
(3.0, 4.0)
```

Inside the function, `p` is an alias for `blank`, so if the function modifies `p`, `blank` changes.

## 15.3 Rectangles

Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions. For example, imagine you are designing a class to represent rectangles. What attributes would you use to specify the location and size of a rectangle? You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal.

There are at least two possibilities:

- You could specify one corner of the rectangle (or the center), the width, and the height.
- You could specify two opposing corners.

At this point it is hard to say whether either is better than the other, so weâĂŹll implement the first one, just as an example.

Here is the class definition:

```
class Rectangle(object):
    """Represents a rectangle.

    attributes: width, height, corner.
    """
```

The docstring lists the attributes: `width` and `height` are numbers; `corner` is a Point object that specifies the lower-left corner.

To represent a rectangle, you have to instantiate a Rectangle object and assign values to the attributes:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

The expression `box.corner.x` means, "Go to the object `box` refers to and select the attribute named `corner`; then go to that object and select the attribute named `x`."

An object that is an attribute of another object is **embedded**.

## 15.4 Instances as return values

Functions can return instances. For example, `find_center` takes a `Rectangle` as an argument and returns a `Point` that contains the coordinates of the center of the `Rectangle`:

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2.0
    p.y = rect.corner.y + rect.height/2.0
    return p
```

Here is an example that passes `box` as an argument as assigns the resulting Point to `center`:

```
>> center = find_center(box)
>> print_point(center)
(50.0, 100.0)
```

## 15.5 Objects are mutable

You can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of `width` and `height`:

```
box.width = box.width + 50
box.height = box.width + 100
```

You can also write functions that modify objects. For example, `grow_rectangle` takes a Rectangle object and two numbers, `dwidth` and `dheight`, and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

Here is an example that demonstrates the effect:

```
>> print box.width
100.0
>> print box.height
200.0
>> grow_rectangle(box, 50, 100)
>> print box.width
150.0
>> print box.height
300.0
```

Inside the function, `rect` is an alias for `box`, so if the function modfies `rect`, `box` changes.

## 15.6 Copying

Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.

Copying an object is often an alternative to aliasing. The `copy` module contains a function called `copy` that can duplicate any object:

```
>> p1 = Point()
>> p1.x = 3.0
>> p1.y = 4.0

>> import copy
>> p2 = copy.copy(p1)
```

`p1` and `p2` contain the same data, but they are not the same Point.

```
>> print_point(p1)
(3.0, 4.0)
>> print_point(p2)
(3.0, 4.0)
>> p1 is p2
False
>> p1 == p2
False
```

The `is` operator indicates that `p1` and `p2` are not the same object, which is what we expected. But you might have expected `==` to yield `True` because these points contain

the same data. In that case, you will be disappointed to learn that for instances, the default behavior of the `==` operator is the same as the `is` operator; it checks object identity, not object equivalence. This behavior can be changedâĂŤweâĂŹll see how later. If you use `copy.copy` to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point.

```
>> box2 = copy.copy(box)
>> box2 is box
False
>> box2.corner is box.corner
True
```

This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects.

For most applications, this is not what you want. In this example, invoking `grow_rectangle` on one of the Rectangles would not affect the other, but invoking `move_rectangle` on either would affect both! This behavior is confusing and error-prone.

Fortunately, the `copy` module contains a method named `deepcopy` that copies not only the object but also the objects it refers to, and the objects *they* refer to, and so on. You will not be surprised to learn that this operation is called a **deep copy**.

```
>> box3 = copy.deepcopy(box)
>> box3 is box
False
>> box3.corner is box.corner
False
```

`box3` and `box` are completely separate objects.

## 15.7 Debugging

When you start working with objects, you are likely to encounter some new exceptions. If you try to access an attribute that doesnâĂŹt exist, you get an `AttributeError`:

```
>> p = Point()
>> print p.z
AttributeError: Point instance has no attribute ¡z¡
```

If you are not sure what type an object is, you can ask:

```
>> type(p)
<type ¡__main__.Point¡>
```

If you are not sure whether an object has a particular attribute, you can use the built-in function `hasattr`:

```
>> hasattr(p, ¡x¡)
True
>> hasattr(p, ¡z¡)
False
```

The first argument can be any object; the second argument is a *string* that contains the name of the attribute.

# 16 Classes and functions

## 16.1 Time

As another example of a user-defined type, we'll define a class called Time that records the time of day. The class definition looks like this:

```
class Time(object):
    """Represents the time of day.

    attributes: hour, minute, second
    """
```

We can create a new Time object and assign attributes for hours, minutes, and seconds:

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

## 16.2 Pure functions

In the next few sections, weâĂŹll write two functions that add time values. They demonstrate two kinds of functions: pure functions and modifiers. They also demonstrate a development plan IâĂŹll call **prototype and patch**, which is a way of tackling a complex problem by starting with a simple prototype and incrementally dealing with the complications.

Here is a simple prototype of `add_time`:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

The function creates a new `Time` object, initializes its attributes, and returns a reference to the new object. This is called a **pure function** because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

To test this function, I'll create two Time objects: `start` contains the start time of a movie, like Monty Python and the Holy Grail, and `duration` contains the run time of the movie, which is one hour 35 minutes.

`add_time` figures out when the movie will be done.

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second =  0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

The result, `10:80:00` might not be what you were hoping for. The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to "carry" the extra seconds into the minute column or the extra minutes into the hour column.

Here's an improved version:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```

Although this function is correct, it is starting to get big. We will see a shorter alternative later.

## 16.3 Modifiers

Sometimes it is useful for a function to modify the objects it gets as parameters. In that case, the changes are visible to the caller. Functions that work this way are called **modifiers**

`increment`, which adds a given number of seconds to a Time object, can be written naturally as a modifier. Here is a rough draft:

```
def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

The first line performs the basic operation; the remainder deals with the special cases we saw before.

Is this function correct? What happens if the parameter `seconds` is much greater than sixty?

In that case, it is not enough to carry once; we have to keep doing it until `time.second` is less than sixty. One solution is to replace the if statements with `while` state-

ments. That would make the function correct, but not very efficient.

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. But modifiers are convenient at times, and functional programs tend to be less efficient.

In general, I recommend that you write pure functions whenever it is reasonable and resort to modifiers only if there is a compelling advantage. This approach might be called a **functional programming style**.

## 16.4 Prototyping versus planning

## 16.5 Debugging

# 17 Classes and methods

# 18 Inheritance

# 19 Case study: Tkinter

# 20 Debugging

# 21 Analysis of Algorithms

# 22 Lumpy

# 23 Plotting

```
-------------------------------------
tight_layout(pad=1.08, h_pad=None, w_pad=None, rect=None)
'''
Automatically adjust subplot parameters to give specified padding.

Parameters:

  pad : float
  padding bet FIGURE edge and the edges of SUBPLOTS,
  as a fraction of the font-size.
  h_pad, w_pad : float
  padding (height/width) between edges of adjacent subplots.
    Defaults to pad_inches.
  rect : if rect is given, it is interpreted as a rectangle
  (left, bottom, right, top) in the normalized figure coordinate
  that the whole subplots area (including labels) will fit into.
  Default is (0, 0, 1, 1).
'''

-------------------------------------
matplotlib.pyplot.subplots_adjust(*args, **kwargs)Ãű
'''
Tune the subplot layout.
call signature::

  subplots_adjust(left=None, bottom=None, right=None, top=None,
                  wspace=None, hspace=None)

Tune the subplot layout via the
:class:`matplotlib.figure.SubplotParams` mechanism.
The parameter meanings (and suggested defaults) are::

  left  = 0.125  #  left side of the subplots
  right = 0.9    #  right side of the subplots
  bottom = 0.1   #  bottom of the subplots
  top = 0.9      #  top of the subplots
  wspace = 0.2   #  width reserved for blank space between subplots
  hspace = 0.2   #  height reserved for white space between subplots

The actual defaults are controlled by the rc file
'''
-------------------------------------
```

# Glossary

| | |
|---|---|
| accumulator | A variable used in a loop to add up or accumulate a result. |
| aliasing | A circumstance where two or more variables refer to the same object. |
| argument | A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function. |
| augmented assignment | A statement that updates the value of a variable using an operator like +=. |
| body | The sequence of statements inside a function definition. |
| composition | Using an expression as part of a larger expression, or a statement as part of a larger statement. |
| data structure | A collection of related values, often organized in lists, dictionaries, tuples, etc. |

| | |
|---|---|
| delimiter | A character or string used to indicate where a string should be split. |
| dot notation | The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name. |
| DSU | Abbreviation of "decorate-sort-undecorate," a pattern that involves building a list of tuples, sorting, and extracting part of the result. |
| element | One of the values in a list (or other sequence), also called items. |
| equivalent | Having the same value. |
| filter | A processing pattern that traverses a list and selects the elements that satisfy some criterion. |
| flow of execution | The order in which statements are executed during a program run. |
| frame | A box in a stack diagram that represents a function call. It contains the local variables and parameters of the function. |
| fruitful function | A function that returns a value. |
| function | A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result. |
| function call | A statement that executes a function. It consists of the function name followed by an argument list. |
| function definition | A statement that creates a new function, specifying its name, parameters, and the statements it executes. |
| function object | A value created by a function definition. The name of the function is a variable that refers to a function object. |
| gather | The operation of assembling a variable-length argument tuple. |
| header | The first line of a function definition. |
| identical | Being the same object (which implies equivalence). |
| import statement | A statement that reads a module file and creates a module object. |
| index | An integer value that indicates an element in a list. |
| list | A sequence of values. |
| list traversal | The sequential accessing of each element in a list. |
| local variable | A variable defined inside a function. A local variable can only be used inside its function. |
| map | A processing pattern that traverses a sequence and performs an operation on each element. |
| mapping | A relationship in which each element of one set corresponds to an element of another set. For example, a list is a mapping from indices to elements. |
| module | A file that contains a collection of related functions and other definitions. |
| module object | A value created by an `import` statement that provides access to the values defined in a module. |
| nested list | A list that is an element of another list. |
| object | Something a variable can refer to. An object has a type and a value. |
| parameter | A name used inside a function to refer to the value passed as an argument. |
| reduce | A processing pattern that traverses a sequence and accumulates the elements into a single result. |
| reference | The association between a variable and its value. |
| return value | The result of a function. If a function call is used as an expression, the return value is the value of the expression. |
| scatter | The operation of treating a sequence as a list of arguments. |
| shape (of a data structure) | A summary of the type, size and composition of a data structure. |
| stack diagram | A graphical representation of a stack of functions, their variables, and the values they refer to. |
| traceback | A list of the functions that are executing, printed when an exception occurs. |
| tuple | An immutable sequence of elements. |
| tuple assignment | An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left. |
| void function | A function that doesn't return a value. |

# 24 Quick reference

## Execution

```
cl> python {options} [-c <command> | -m <module
    > | file - ] {arguments}
    -h  # print fill list of options and
    summary help
    -m  # specify module to run as main script

cl> export PYTHONPATH="${PYTHONPATH}:/new/path"
>>> import sys
>>> sys.path
$
```

(because lstlisting doesn't know what to do with dollar signs).

## Debugging

```
import pdb
pdb.set_trace() #equivalent of 'STOP' in IDL
cl> python code.py
...
(Pdb) continue
```

Type 'continue' at the end to keep going. Type 'q' to quit. Can also print variables, types, etc. in this mode.

## Modules

```
sys   #
import sys.argv # variable with list of strings
     (script name + arguments)
```

## Strings

```
>>> s1 = "/home/users/laurel/"
>>> s2 = "file.dat"
>>> s1 + s2
  "/home/users/laurel/file.dat"
>>> s2[0:5]
  'file.'
>>> s2[:5]
  'file.'
>>> s2[-5:]
  'e.dat'
>>> len(s2)
  8
```

## Operators

### Mathematical

+,-,*,/,**,+=,-=,*=,/=,++,--

### Relational

==, !=, >, >=, <, <=

### Boolean

Bitwise:

```
&  # and
|  # or
^  # xor
!  # not
```

Logical:

```
and, or, not (?)
```

## conditionals

```
if (condition):
    statements
elif (condition):
    statements
else:
    statements
    pass # need to have 'something' here. ???
if (condition):
    break
else:
    continue
```

## Looping

Python is not inclusive!

```
for i in list:
for i in range(len(list)):
for i in range(0, 10, 0.1)

for i in range([start,]stop[, step])
```

(by default, start = 0 and step = 1)

```
>>> root = "output."
>>> for i in range(1,101):
    ... outfile = root + repr(i).zfill(3) + '.
    txt'
    ... print outfile

output.001.txt
...
output.100.txt
```

repr(i) returns a string and zfill(3) pads with leading zeros.

```
while (condition):
    statements
```

## I/O

Simple output:

```
print 'characters', string, variable
```

Simple reading from terminal and file:

```
terminal:
s = raw_input('prompt')
```

Higher level file reading routines:

```
data = numpy.loadtxt(file,
dtype=[('name','a12'),('ra',f4),('dec','f4')])
data = astropy.io.ascii(file)
```

## Type changes

```
str(x)      # num to string
abs(x)      # absolute value
int(x)      # float to integer (truncated)
long(x)     # long data type
float(x)    # takes string or int, NOT array
```

## Math

```
import math
# Use the following as math.funcname(x), unless
    specified otherwise
pow(x,y)   # x^y
sqrt
log
log10
exp
fabs
sin, cos, tan, asin, acos, atan
atan2(y,x)  # vector to (x,y) makes this angle
    with the positive x-axis. Returns values
    between -pi and +pi
radians(x) # degrees --> radians
degrees(x) # radians --> degrees

from scipy.interpolate import interp1d
interp1d(x, y, 'cubic')  # x is data, y = f(x)
from scipy.interpolate import interp2d
interp2d(x, y, z, 'cubic')  # x and y are data,
    z = f(x,y)
```

## Lists

```
>>> x = range(5)
    [0,1,2,3,4]
>>> len(x)
    5
>>> x.append(5)
    [0,1,2,3,4,5]

>>> A = [1,2,3]
  [1,2,3]
>>> A*2  # Get A back twice
  [1,2,3,1,2,3]
```

## numpy arrays

Compare to list example above:

```
import numpy as np
>>> B = np.array([1,2,3])
  [1 2 3]
>>> B+2  # Element by element
  [3 4 5]
>>> B*2
  [2 4 6]

A.shape      # Dimensions (rows, columns)
A.ndim       # Number of dimensions
A.size       # Number of elements
A.astype(float)  # Change to type float

np.log(A)    # Natural log
np.log10(A)  # log base 10
np.atan2(A)  # array of angles between -pi and
    +pi

np.array(object, dtype=None)
np.zeros([n,m])  # n=rows, m=columns
np.arange([start,] stop, [step,] dtype=None)
np.linspace([start,] stop, [step,] dtype=None)
     # num_steps = 50 by default
np.logspace
np.ndarray(...) # two-dimensional array
np.argmax(x) # returns INDEX of max value of x
    array
np.where(condition) --> array
np.roll(A, n, axis=None)  # Default: A is
    flattened, shifted, then restored (~IDL's
    SHIFT)
np.reshape(A, shape) # shape = int or (rows,
    cols)
np.stack
np.append(A, values, axis=None)
np.broadcast_to(array, shape)
np.repeat(array, num_repeats, axis=...)

A.flatten()  # Returns 1D array


# np.funcname(array, axis=None, dtype=None) ~ A
    .funcname(...)

# Statistics
sum, max, min, mean, std
np.std(A, axis=None); A.std()
A.var()  # Normalized with N (not N-1)
np.median(A, axis=None)
     axis - along which median is computed. Get
     array back, or if no
     axis is specified, get a single number back
     .
     A = [[1 2 3]
          [4 5 6]
          [7 8 9]]

     np.median(A) --> 5.0
     np.median(A, axis=0) --> [4 5 6]
     np.median(A, axis=1) --> [2 5 8]
     np.median(A, axis=2) --> Error

A = complex(3.2, 5.4)
A = (3.2 + 5.4j)  # (equivalent)
A.imag --> 5.4
A.real --> 3.2  # return value is same type as
    input. If input is complex, returns a float

np.cross(x,y)
np.dot(x,y)

A = np.matrix(A)
A = np.mat(A)  # Difference?
B = (np.matrix(B)).reshape(3,1)
B*A  # Matrix multiplication

# Fourier transform
f = np.fft.fft(A, n=None, axis=-1, norm=None)
```

## Functions

```
def lag(m, *v):      # v is a parameter
var = lag(m, "hi")

def lag(m, v=None): # v is a keyword
var = lag(m, v="hi")

if v: ...
if not v: ...
```

## FITS files

```
import glob
fls = glob.glob('*.fits') # list of filenames

from astropy.io import fits
hdu = fits.open('name.fits') # Create list
    object.
hdu.info()
info = hdu[0].header
image = hdu[0].data # type(image) --> numpy.
    ndarray
image.shape # shape (dimensions) of numpy.
    ndarray
hdu.close() # Always do this
```

## Dynamic memory allocation

```
# structures (dictionaries)
var={'name':'test', ra:1.}
print var['name'], var['test']
dicname = { 'A':[1,2,3], 'B':[2,4,6], 'C'
    :[3,9,15] }
print dicname
# dictionaries are indexed by keys (A, B, C),
    rather than numbers.
# Keys can be any immutable type. Can't use
    lists, since they can be modified.
profile = {'density':rho_r, 'mass':mass, ...}
my_density = np.array(profile['density'])

# structure arrays
sarray = np.zeros(nelem,
 dtype=[('name','a12'),('ra','f4'),('dec','f4')
    ])
sarray=np.zeros(nelem,
 dtype={'names': ('name', 'ra', 'dec'),
  'formats': ('S12','f4','f4')})
```

## Classes

```
# To allow import:
def main():
    # program statements
    print "hello world."
if __name__=="__main__": # flexible way of
    running routines
    main()
```

## Plotting

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(20,10))  # Inches
fig.suptitle|xlabel|ylabel(
    'string',
    fontsize=14,
    fontweight='bold',
    color='red')
for i in range(r*c):
    ax = fig.add_subplot(r,c,i+1)

fig, ax = plt.subplots()
fig,((ax1,ax2),(ax3,ax4)) = plt.subplots(2,2)
axes = (ax1,ax1,ax1,ax1)
for ax in axes:
    ax.plot(...)
```

### Adjust space between and around plots

```
fig.set_size_inches(width,height)  # Not same
    as figsize...

# Units: fraction of fontsize
plt|fig.tight_layout(pad=0.4, #around fig
    border
    w_pad=0.5, h_pad=1.0  #between subplots
    )
#For saving figure to disk
plt.savefig("im.png", bbox_inches="tight", dpi
    =300)
```

### Individual axes

```
ax.set_xlim(left=xmin,right=xmax)
ax.set_ylim(bottom=ymin,top=ymax)
ax.set_xscale('linear'|'log')
ax.minorticks_on()
ax.tick_params( # Formatting appearence
    axis=['both'|'x'|'y'],
    which=['major'|'minor'|'both'],
    direction=['in'|'out'|'both'],
    length=6, width=2, colors='r',
    labelsize=12, labelcolor='k',
    pad=5, # bet. label \& tick
    bottom, top, left, right:[bool|'on'|'off'],
    label[bottom|top|left|right]:['on'|'off']
    )
ax.set_xticks(
    np.arange(start,stop,step),
    minor=False #default
    )
ax.set_xticklabels(
    labels,
    fontdict=None,
    minor=False, #default
    rotation=angle|'vertical'|'horizontal',
    fontsize=10, #points
    fontstyle='normal'|'italic'|'oblique',
    ha=['center'|'right'|'left'],
    va=['center'|'top'|'bottom'|'baseline']
    )
ax.ticklabel_format(
    style=['scientific'|'plain'],
    scilimits=(m,n), #integers
    axis=['x'|'y'|'both'],
    )
ax.set_xlabel(
    'labelname',
    fontsize = 14,
    color='#eeefff',
    labelpad=None, #[points] bet. label \& x-
    axis
    )
ax.axis(
    'equal' # same numerical range for x and y
    'scaled' # equal scaling by changing box
    dimensions
    'square' # x1-x0 = y1-y0
    'tight' # squeeze out extra blank space
    around data.
)
ax.fill(), ax.fill_between()
ax.contour()
ax.clabel(cs, v, **kwargs)
# Label a contour plot
# only label contours listed in v.
```

### Scatter plots

```
p = ax.scatter(x, y,
    c=3D_data_array, #colors
    vmin=min(z), vmax=max(z),
    s=1, #size
    cmap=cm,
    lw=0 #no linewidth (border)
    )
```

### Colorbar

```
fig, ax = plt.subplots()
m = ax.scatter(data, ...)
cax = fig.add_axes([left,bottom,width,height])
plt|fig.colorbar(m, cax=cax,
    orientation='horizontal' #color change/
    labels, not direction of bar itself
    )
```

### Put two plots on the same axes

```
ax1.plot(x,y1,'r')
ax2 = ax1.twinx()
ax2.plot(x,y2,'g')
for tl in ax2.get_yticklabels():
    tl.set_color('g')
ax.tick_params(labelsize="small") # size of
    numbers on each tick mark, not axis titles
```

x and y limits: order matters! Put after set x ticks.

### Imaging

```
ax.imshow(hdu.data, cmap='gray')
cube = np.stack([im1, im2, im3], axis=2)
np.power(image, 0.1)
```

### Legend

```
ax.plot(x,y1, 'k—', label='line 1')
ax.plot(x,y2, 'k:', label='line 2')
ax.plot(x,y3, 'k', label='line 3')
legend = ax.legend(loc='upper center',
                   = 'lower left',
              shadow = True)
```

### Misc

```
ax.streamplot()  # Draw streamlines of a vector
    flow!
```