Google's Python Class

Welcome to Google's Python Class -- this is a free class for people with a little bit of programming experience who want to learn Python. The class includes written materials, lecture videos, and lots of code exercises to practice Python coding. These materials are used within Google to introduce Python to people who have just a little programming experience. The first exercises work on basic Python concepts like strings and lists, building up to the later exercises which are full programs dealing with text files, processes, and http connections. The class is geared for people who have a little bit of programming experience in some language, enough to know what a "variable" or "if statement" is. Beyond that, you do not need to be an expert programmer to use this material.

To get started, the Python sections are linked at the left -- Python Set Up to get Python installed on your machine, Python Introduction for an introduction to the language, and then Python Strings starts the coding material, leading to the first exercise. The end of each written section includes a link to the code exercise for that section's material. The lecture videos parallel the written materials, introducing Python, then strings, then first exercises, and so on. At Google, all this material makes up an intensive 2-day class, so the videos are organized as the day-1 and day-2 sections.

This material was created by Nick Parlante working in the engEDU group at Google. Special thanks for the help from my Google colleagues John Cox, Steve Glassman, Piotr Kaminksi, and Antoine Picard. And finally thanks to Google and my director Maggie Johnson for the enlightened generosity to put these materials out on the internet for free under the under the Creative Commons Attribution 2.5 license -- share and enjoy!

Tip: Check out the Python Google Code University Forum to ask and answer questions.

Last updated December 13, 2012.

Python Set Up

This page explains how to set up Python on a machine so you can run and edit Python programs, and links to the exercise code to download. You can do this before starting the class, or you can leave it until you've gotten far enough in the class that you want to write some code. The Google Python Class uses a simple, standard Python installation, although more complex strategies are possible. Python is free and open source, available for all operating systems from python.org. In particular we want a Python install where you can do two things:

- Run an existing python program, such as hello.py
- Run the Python interpreter interactively, so you can type code right at it

Both of the above are done quite a lot in the lecture videos, and it's definitely something you need to be able to do to solve the exercises.

Download Google Python Exercises

As a first step, download the google-python-exercises.zip file and unzip it someplace where you can work on it. The resulting google-python-exercises directory contains many different python code exercises you can work on. In particular, google-python-exercises contains a simple hello.py file you can use in the next step to check that Python is working on your machine. Below are Python instructions for Windows and all other operation systems:

Python on Linux, Mac OS X, etc.

Most operating systems other than Windows already have Python installed by default. To check that Python is installed, open a command line (typically by running the "Terminal" program), and cd to the google-python-exercises directory. Try the following to run the hello.py program (what you type is shown in bold):

~/google-python-exercises$ **python hello.py**
Hello World
~/google-python-exercises$ **python hello.py Alice**
Hello Alice

If python is not installed, see the Python.org download page. To run the Python interpreter interactively, just type "python" in the terminal:

~/google-python-exercises$ **python**
Python 2.5.2 (r252:60911, Feb 22 2008, 07:57:53)
[GCC 4.0.1 (Apple Computer, Inc. build 5363)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> **1 + 1**
2
>>> **you can type expressions here .. use ctrl-d to exit**

For Google's Python Class, you want a python version that is 2.4 or later, and avoiding the 3.x versions for now is probably best.

Execute Bit (optional)

The commands above are the simplest way to run python programs. If the "execute bit" is set on a .py file, it can be run by name without having to type "python" first. Set the execute bit with the "chmod" command like this:

~/google-python-exercises$ **chmod +x hello.py**
~/google-python-exercises$ **./hello.py** ## now can run it as ./hello.py
Hello World

Python on Windows

Doing a basic Python install on Windows is easy:

- Go to the python.org download page, select a version such as 2.6. Google's Python Class should work with any version 2.4 or later, and avoiding the 3.x versions for now is probably best.
- Run the Python installer, taking all the defaults. This will install Python in the root directory and set up some file associations.

1. With Python installed, open a command prompt (Accessories > Command Prompt, or type 'cmd' into the run dialog). Cd to the google-python-exercises directory (from unzipping google-python-exercises.zip). You should be able to run the hello.py python program by typing "python" followed by "hello.py" (what you type is shown in bold):C:\google-python-exercises> **python hello.py**
   Hello World
   C:\google-python-exercises> **python hello.py Alice**
   Hello Alice

   If this works, Python is installed. Otherwise, see Python Windows FAQ for help.To run the Python interpreter interactively, select the Run... command from the Start menu, and type "python" -- this will launch Python interactively in its own window. On Windows, use Ctrl-Z to exit (on all other operating systems it's Ctrl-D to exit).In the lecture videos, we generally run the Python programs with commands like "./hello.py". On Windows, it's simplest to use the "python hello.py" form.**Editing Python (all operating systems)**A Python program is just a text file that you edit directly. As above, you should have a command line open, where you can type "python hello.py Alice" to run whatever exercise you are working on. At the command line prompt, just hit the up-arrow key to recall previously typed commands, so it's easy to run previous commands without retyping them.You want a text editor with a little understanding of code and indentation. There are many good free ones:

- Windows -- **do not use Notepad or Wordpad**. Try the free and open source Notepad++ or the free and open source JEdit
- Mac -- The built in TextEdit works, but not very well. Try the free TextWrangler or the free and open source JEdit
- Linux -- any unix text editor is fine, or try the above JEdit.

1. To edit Python, we advocate the strategy that when you hit the tab key, your editor inserts spaces rather than a real tab character. All our files use 2-spaces as the indent, and 4-spaces is another popular choice. It's also handy if the editor will "auto indent" so when you hit return, the new line starts with the same indentation as the previous line. We also recommend saving your files with the unix line-ending convention, since that's how the various starter files are set up. If running hello.py gives the error "Unknown option: -", the file may have the wrong line-ending. Here are the preferences to set for common editors to treat tabs and line-endings correctly for Python:**Editor Settings**

- Windows Notepad++ -- Tabs: Settings > Preferences > Edit Components > Tab settings, and Settings > Preferences > MISC for auto-indent. Line endings: Format > Convert, set to Unix.
- JEdit (any OS) -- Line endings: Little 'U' 'W' 'M' on status bar, set it to 'U' (i.e. Unix line-endings)
- Windows Notepad or Wordpad -- do not use
- Mac TextWrangler -- Tabs: Preference button at the top of the window, check Auto Expand Tabs. Can set the default in Defaults > Auto-Expand Tabs and Auto-indent. Line endings: little control at the bottom of each window, set it to Unix
- Mac TextEdit -- do not use
- Unix pico -- Tabs: Esc-q toggles tab mode, Esc-i to turns on auto-indent mode
- Unix emacs -- Tabs: manually set tabs-inserts-spaces mode: M-x set-variable(return) indent-tabs-mode(return) nil

1. One of the advantages of Python is that it makes it easy to type a little code and quickly see what it does. In class, we want a work setup that matches that .. a text text editor working on the current file.py, and a separate command line window where you can just hit the up-arrow key to run file.py and see what it does. (Teaching philosophy aside: the interpreter is great for little experiments, as shown throughout the lectures. However, the exercises are structured as Python files that students edit. Since being able to write Python programs is the ultimate goal, I felt it was best to be in that mode the whole time, using the interpreter just for little experiments.)**Quick Python Style**To try out your editor, edit the the hello.py program. Change the word "Hello" in the code to the word "Howdy" (you don't need to understand all the other Python code in there ... we'll explain it all in class). Save your edits and run the program to see its new output. Try adding a "print 'yay!'" just below the existing print and with the same indentation. Try running the program, to see that your edits work correctly. For class we want an edit/run workflow that allows you to switch between editing and running easily.**Editing Check**

Python Introduction

Prelude

Welcome to Google's Python online tutorial. It is based on the introductory Python course offered internally. Originally created during the Python 2.4 days, we've tried to keep the content universal and exercises relevant, even for newer releases. As mentioned on the setup page, this material covers Python 2. While we recommend "avoiding" Python 3 for now, recognize that it is the future, as all new features are only going there. The good news is that developers learning either version can pick up the other without too much difficulty. If you want to know more about choosing Python 2 vs. 3, check out this post.

We strongly recommend you follow along with the companion videos throughout the course, starting with the first one. If you're seeking a companion MOOC course, try the ones from Udacity and Coursera (intro to programming [beginners] or intro to Python), and if you're looking for a companion book to your learning, regardless of your Python skill level, check out these reading lists. Finally, if you're seeking self-paced online learning *without* watching videos, try the ones listed towards the end of this post — each feature learning content as well as a Python interactive interpreter you can practice with. What's this "interpreter" we mention? You'll find out in the next section!

Language Introduction

Python is a dynamic, interpreted (bytecode-compiled) language. There are no type declarations of variables, parameters, functions, or methods in source code. This makes the code short and flexible, and you lose the compile-time type checking of the source code. Python tracks the types of all values at runtime and flags code that does not make sense as it runs.

An excellent way to see how Python code works is to run the Python interpreter and type code right into it. If you ever have a question like, "What happens if I add an int to a list?" Just typing it into the Python interpreter is a fast and likely the best way to see what happens. (See below to see what really happens!)

$ python   ## Run the Python interpreterPythondefault20080704 on linux2

"help""copyright""credits""license" more information>>>## set a variable in this interpreter session>>>## entering an expression prints its value>>>>>>## 'a' can hold a string just as well>>>>>>## call the len() function on a string>>>## try something that doesn't workTracebackmost recent call TypeError cannot concatenate 'str''int' objects

>>>## probably what you really wanted'hi2'>>>## try something else that doesn't workTracebackmost recent call NameError'foo'defined>>>## type CTRL-d to exit (CTRL-z in Windows/DOS terminal)

As you can see above, it's easy to experiment with variables and operators. Also, the interpreter throws, or "raises" in Python parlance, a runtime error if the code tries to read a variable that has not been assigned a value. Like C++ and Java, Python is case sensitive so "a" and "A" are different variables. The end of a line marks the end of a statement, so unlike C++ and Java, Python does not require a semicolon at the end of each statement. Comments begin with a '#' and extend to the end of the line.

Python source code

Python source files use the ".py" extension and are called "modules." With a Python module hello.py, the easiest way to run it is with the shell command "python hello.py Alice" which calls the Python interpreter to execute the code in hello.py, passing it the command line argument "Alice". See the official docs page on all the different options you have when running Python from the command-line.

Here's a very simple hello.py program (notice that blocks of code are delimited strictly using indentation rather than curly braces — more on this later!):

#!/usr/bin/env python# import modules used here -- sys is a very standard oneimport# Gather our code in a main() functionprint'Hello there'# Command line args are in sys.argv[1], sys.argv[2] ...# sys.argv[0] is the script name itself and can be ignored# Standard boilerplate to call the main() function to begin# the program. __name__ '__main__'

Running this program from the command line looks like:

$ python hello.py Guido
Hello there Guido
$ ./hello.py Alice ## without needing 'python' first (Unix)
Hello there Alice

Imports, Command-line arguments, and len()

The outermost statements in a Python file, or "module", do its one-time setup — those statements run from top to bottom the first time the module is imported somewhere, setting up its variables and functions. A Python module can be run directly — as above "python hello.py Bob" — or it can be imported and used by some other module. When a Python file is run directly, the special variable "__name__" is set to "__main__". Therefore, it's common to have the boilerplate if __name__ ==... shown above to call a main() function when the module is run directly, but not when the module is imported by some other module.

In a standard Python program, the list sys.argv contains the command-line arguments in the standard way with sys.argv[0] being the program itself, sys.argv[1] the first argument, and so on. If you know about argc, or the number of arguments, you can simply request this value from Python with len(sys.argv), just like we did in the interactive interpreter code above when requesting the length of a string. In general, len() can tell you how long a string is, the number of elements in lists and tuples (another array-like data structure), and the number of key-value pairs in a dictionary.

User-defined Functions

Functions in Python are defined like this:

Defines a "repeat" function that takes 2 arguments. repeat exclaim***

    Returns the string 's' repeated 3 times.

    If exclaim is true, add exclamation marks.

    ***

result # can also use "s * 3" which is faster (Why?) exclaim

      result result '!!!'return result

Notice also how the lines that make up the function or if-statement are grouped by all having the same level of indentation. We also presented 2 different ways to repeat strings, using the + operator which is more user-friendly, but * also works because it's Python's "repeat" operator, meaning that '-' * 10 gives '----------', a neat way to create an onscreen "line." In the code comment, we hinted that * works faster than +, the reason being that * calculates the size of the resulting object once whereas with +, that calculation is made each time + is called. Both + and * are called "overloaded" operators because they mean different things for numbers vs. for strings (and other data types).

The def keyword defines the function with its parameters within parentheses and its code indented. The first line of a function can be a documentation string ("docstring") that describes what the function does. The docstring can be a single line, or a multi-line description as in the example above. (Yes, those are "triple quotes," a feature unique to Python!) Variables defined in the function are local to that function, so the "result" in the above function is separate from a "result" variable in another function. The return statement can take an argument, in which case that is the value returned to the caller.

Here is code that calls the above repeat() function, printing what it returns:

print repeat'Yay'False## YayYayYayprint repeat'Woo Hoo'## Woo HooWoo HooWoo Hoo!!!

At run time, functions must be defined by the execution of a "def" before they are called. It's typical to def a main() function towards the bottom of the file with the functions it calls above it.

Indentation

One unusual Python feature is that the whitespace indentation of a piece of code affects its meaning. A logical block of statements such as the ones that make up a function should all have the same indentation, set in from the indentation of their parent function or "if" or whatever. If one of the lines in a group has a different indentation, it is flagged as a syntax error.

Python's use of whitespace feels a little strange at first, but it's logical and I found I got used to it very quickly. Avoid using TABs as they greatly complicate the indentation scheme (not to mention TABs may mean different things on different platforms). Set your editor to insert spaces instead of TABs for Python code.

A common question beginners ask is, "How many spaces should I indent?" According to the official Python style guide (PEP 8), you should indent with 4 spaces. (Fun fact: Google's internal style guideline dictates indenting by 2 spaces!)

Code Checked at Runtime

Python does very little checking at compile time, deferring almost all type, name, etc. checks on each line until that line runs. Suppose the above main() calls repeat() like this:

'Guido'print repeeeet'!!!'print repeat

The if-statement contains an obvious error, where the repeat() function is accidentally typed in as repeeeet(). The funny thing in Python ... this code compiles and runs fine so long as the name at runtime is not 'Guido'. Only when a run actually tries to execute the repeeeet() will it notice that there is no such function and raise an error. This just means that when you first run a Python program, some of the first errors you see will be simple typos like this. This is one area where languages with a more verbose type system, like Java, have an advantage ... they can catch such errors at compile time (but of course you have to maintain all that type information ... it's a tradeoff).

Variable Names

Since Python variables don't have any type spelled out in the source code, it's extra helpful to give meaningful names to your variables to remind yourself of what's going on. So use "name" if it's a single name, and "names" if it's a list of names, and "tuples" if it's a list of tuples. Many basic Python errors result from forgetting what type of value is in each variable, so use your variable names (all you have really) to help keep things straight.

As far as actual naming goes, some languages prefer underscored_parts for variable names made up of "more than one word," but other languages prefer camelCasing. In general, Python prefers the underscore method but guides developers to defer to camelCasing if integrating into existing Python code that already uses that style. Readability counts. Read more in the section on naming conventions in PEP 8.

As you can guess, keywords like 'print' and 'while' cannot be used as variable names — you'll get a syntax error if you do. However, be careful not to use built-ins as variable names. For example, while 'str' and 'list' may seem like good names, you'd be overriding those system variables. Built-ins are not keywords and thus, are susceptible to inadvertent use by new Python developers.

More on Modules and their Namespaces

Suppose you've got a module "binky.py" which contains a "def foo()". The fully qualified name of that foo function is "binky.foo". In this way, various Python modules can name their functions and variables whatever they want, and the variable names won't conflict — module1.foo is different from module2.foo. In the Python vocabulary, we'd say that binky, module1, and module2 each have their own "namespaces," which as you can guess are variable name-to-object bindings.

For example, we have the standard "sys" module that contains some standard system facilities, like the argv list, and exit() function. With the statement "import sys" you can can then access the definitions in the sys module and makes them available by their fully-qualified name, e.g. sys.exit(). (Yes, 'sys' has a namespace too!)

import# Now can refer to sys.xxx facilities

There is another import form that looks like this: "from sys import argv, exit". That makes argv and exit() available by their short names; however, we recommend the original form with the fully-qualified names because it's a lot easier to determine where a function or attribute came from.

There are many modules and packages which are bundled with a standard installation of the Python interpreter, so you don't have do anything extra to use them. These are collectively known as the "Python Standard Library." Commonly used modules/packages include:

- sys — access to exit(), argv, stdin, stdout, ...
- re — regular expressions
- os — operating system interface, file system

You can find the documentation of all the Standard Library modules and packages at http://docs.python.org/library.

Online help, help(), and dir()

There are a variety ways to get help for Python.

- Do a Google search, starting with the word "python", like "python list" or "python string lowercase". The first hit is often the answer. This technique seems to work better for Python than it does for other languages for some reason.
- The official Python docs site — docs.python.org — has high quality docs. Nonetheless, I often find a Google search of a couple words to be quicker.
- There is also an official Tutor mailing list specifically designed for those who are new to Python and/or programming!
- Many questions (and answers) can be found on StackOverflow and Quora.
- Use the help() and dir() functions (see below).

Inside the Python interpreter, the help() function pulls up documentation strings for various modules, functions, and methods. These doc strings are similar to Java's javadoc. The dir() function tells you what the attributes of an object are. Below are some ways to call help() and dir() from the interpreter:

- help(len) — help string for the built-in len() function; note that it's "len" not "len()", which is a **call** to the function, which we don't want
- help(sys) — help string for the sys module (must do an import sys first)
- dir(sys) — dir() is like help() but just gives a quick list of its defined symbols, or "attributes"
- help(sys.exit) — help string for the exit() function in the sys module
- help('xyz'.split) — help string for the split() method for string objects. You can call help() with that object itself or an **example** of that object, plus its attribute. For example, calling help('xyz'.split) is the same as calling help(str.split).
- help(list) — help string for list objects
- dir(list) — displays list object attributes, including its methods
- help(list.append) — help string for the append() method for list objects

Python Strings

Python has a built-in string class named "str" with many handy features (there is an older module named "string" which you should not use). String literals can be enclosed by either double or single quotes, although single quotes are more commonly used. Backslash escapes work the usual way within both single and double quoted literals -- e.g. \n \' \". A double quoted string literal can contain single quotes without any fuss (e.g. "I didn't do it") and likewise single quoted string can contain double quotes. A string literal can span multiple lines, but there must be a backslash \ at the end of each line to escape the newline. String literals inside triple quotes, """ or ''', can multiple lines of text.

Python strings are "immutable" which means they cannot be changed after they are created (Java strings also use this immutable style). Since strings can't be changed, we construct *new* strings as we go to represent computed values. So for example the expression ('hello' + 'there') takes in the 2 strings 'hello' and 'there' and builds a new string 'hellothere'.

Characters in a string can be accessed using the standard [ ] syntax, and like Java and C++, Python uses zero-based indexing, so if str is 'hello' str[1] is 'e'. If the index is out of bounds for the string, Python raises an error. The Python style (unlike Perl) is to halt if it can't tell what to do, rather than just make up a default value. The handy "slice" syntax (below) also works to extract any substring from a string. The len(string) function returns the length of a string. The [ ] syntax and the len() function actually work on any sequence type -- strings, lists, etc.. Python tries to make its operations work consistently across different types. Python newbie gotcha: don't use "len" as a variable name to avoid blocking out the len() function. The '+' operator can concatenate two strings. Notice in the code below that variables are not pre-declared -- just assign to them and go.

printprintprint' there'## hi there

Unlike Java, the '+' does not automatically convert numbers or other types to string form. The str() function converts values to a string form so they can be combined with other strings.

##text = 'The value of pi is ' + pi   ## NO, does not work'The value of pi is '## yes

For numbers, the standard operators, +, /, * work in the usual way. There is no ++ operator, but +=, -=, etc. work. If you want integer division, it is most correct to use 2 slashes -- e.g. 6 // 5 is 1 (previous to python 3000, a single / does int division with ints anyway, but moving forward // is the preferred way to indicate that you want int division.)

The "print" operator prints out one or more python items followed by a newline (leave a trailing comma at the end of the items to inhibit the newline). A "raw" string literal is prefixed by an 'r' and passes all the chars through without special treatment of backslashes, so r'x\nx' evaluates to the length-4 string 'x\nx'. A 'u' prefix allows you to write a unicode string literal (Python has lots of other unicode support features -- see the docs below).

'this\t\n and that'print## this\t\n and that

multi """It was the best of times.

  It was the worst of times."""

String Methods

Here are some of the most common string methods. A method is like a function, but it runs "on" an object. If the variable s is a string, then the code s.lower() runs the lower() method on that string object and returns the result (this idea of a method running on an object is one of the basic ideas that make up Object Oriented Programming, OOP). Here are some of the most common string methods:

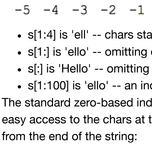
- s.lower(), s.upper() -- returns the lowercase or uppercase version of the string
- s.strip() -- returns a string with whitespace removed from the start and end
- s.isalpha()/s.isdigit()/s.isspace()... -- tests if all the string chars are in the various character classes
- s.startswith('other'), s.endswith('other') -- tests if the string starts or ends with the given other string
- s.find('other') -- searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found
- s.replace('old', 'new') -- returns a string where all occurrences of 'old' have been replaced by 'new'
- s.split('delim') -- returns a list of substrings separated by the given delimiter. The delimiter is not a regular expression, it's just text. 'aaa,bbb,ccc'.split(',') -> ['aaa', 'bbb', 'ccc']. As a convenient special case s.split() (with no arguments) splits on all whitespace chars.
- s.join(list) -- opposite of split(), joins the elements in the given list together using the string as the delimiter. e.g. '---'.join(['aaa', 'bbb', 'ccc']) -> aaa---bbb---ccc

A google search for "python str" should lead you to the official python.org string methods which lists all the str methods.

Python does not have a separate character type. Instead an expression like s[8] returns a string-length-1 containing the character. With that string-length-1, the operators ==, <=, ... all work as you would expect, so mostly you don't need to know that Python does not have a separate scalar "char" type.

String Slices

The "slice" syntax is a handy way to refer to sub-parts of sequences -- typically strings and lists. The slice s[start:end] is the elements beginning at start and extending up to but not including end. the Suppose we have s = "Hello"



- s[1:4] is 'ell' -- chars starting at index 1 and extending up to but not including index 4
- s[1:] is 'ello' -- omitting either index defaults to the start or end of the string
- s[:] is 'Hello' -- omitting both always gives us a copy of the whole thing (this is the pythonic way to copy a sequence like a string or list)
- s[1:100] is 'ello' -- an index that is too big is truncated down to the string length

The standard zero-based index numbers give easy access to chars near the start of the string. As an alternative, Python uses negative numbers to give easy access to the chars at the end of the string: s[-1] is the last char 'o', s[-2] is 'l' the next-to-last char, and so on. Negative index numbers count back from the end of the string:

- s[-1] is 'o' -- last char (1st from the end)
- s[-4] is 'e' -- 4th from the end
- s[:-3] is 'He' -- going up to but not including the last 3 chars.
- s[-3:] is 'llo' -- starting with the 3rd char from the end and extending to the end of the string.

It is a neat truism of slices that for any index n, s[:n] + s[n:] == s. This works even for n negative or out of bounds. Or put another way s[:n] and s[n:] always partition the string into two string parts, conserving all the characters. As we'll see in the list section later, slices work with lists too.

String %

Python has a printf()-like facility to put together a string. The % operator takes a printf-type format string on the left (%d int, %s string, %f/%g floating point), and the matching values in a tuple on the right (a tuple is made of values separated by commas, typically grouped inside parenthesis):

% operator"%d little pigs come out or I'll %s and %s and %s"'huff''puff''blow down'

The above line is kind of long -- suppose you want to break it into separate lines. You cannot just split the line after the '%' as you might in other languages, since by default Python treats each line as a separate statement (on the plus side, this is why we don't need to type semi-colons on each line). To fix this, enclose the whole expression in an outer set of parenthesis -- then the expression is allowed to span multiple lines. This code-across-lines technique works with the various grouping constructs detailed below: ( ), [ ], { }.

add parens to make the long-line work:"%d little pigs come out or I'll %s and %s and %s"'huff''puff''blow down'

i18n Strings (Unicode)

Regular Python strings are *not* unicode, they are just plain bytes. To create a unicode string, use the 'u' prefix on the string literal:

ustring 'A unicode \u018e string \xf1' ustring

u'A unicode \u018e string \xf1'

A unicode string is a different type of object from regular "str" string, but the unicode string is compatible (they share the common superclass "basestring"), and the various libraries such as regular expressions work correctly if passed a unicode string instead of a regular string.

To convert a unicode string to bytes with an encoding such as 'utf-8', call the ustring.encode('utf-8') method on the unicode string. Going the other direction, the unicode(s, encoding) function converts encoded plain bytes to a unicode string:

(ustring from above contains a unicode string) ustringencode'utf-8''A unicode \xc6\x8e string \xc3\xb1'## bytes of utf-8 encoding unicode'utf-8'## Convert bytes back to a unicode string ustring   ## It's the same as the original, yay!

The built-in print does not work fully with unicode strings. You can encode() first to print in utf-8 or whatever. In the file-reading section, there's an example that shows how to open a text file with some encoding and read out unicode strings. Note that unicode handling is one area where Python 3000 is significantly cleaned up vs. Python 2.x behavior described here.

If Statement

Python does not use { } to enclose blocks of code for if/loops/function etc.. Instead, Python uses the colon (:) and indentation/whitespace to group statements. The boolean test for an if does not need to be in parenthesis (big difference from C++/Java), and it can have *elif* and *else* clauses (mnemonic: the word "elif" is the same length as the word "else").

Any value can be used as an if-test. The "zero" values all count as false: None, 0, empty string, empty list, empty dictionary. There is also a Boolean type with two values: True and False (converted to an int, these are 1 and 0). Python has the usual comparison operations: ==, !=, <, <=, >, >=. Unlike Java and C, == is overloaded to work correctly with strings. The boolean operators are the spelled out words *and*, *or*, *not* (Python does not use the C-style && || !). Here's what the code might look like for a policeman pulling over a speeder -- notice how each block of then/else statements starts with a : and the statements are grouped by their indentation:

speed >=print'License and registration please''terrible' speed >=print'You have the right to remain silent.''bad' speed >=print"I'm going to have to write you a ticket."

    write_ticketprint"Let's try to keep it under 80 ok?"

I find that omitting the ":" is my most common syntax mistake when typing in the above sort of code, probably since that's an additional thing to type vs. my C++/Java habits. Also, don't put the boolean test in parens -- that's a C/Java habit. If the code is short, you can put the code on the same line after ":", like this (this applies to functions, loops, etc. also), although some people feel it's more readable to space things out on separate lines.

speed >=print'You are so busted'print'Have a nice day'

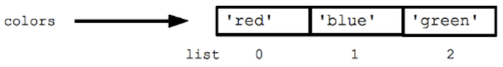
Exercise: string1.py

To practice the material in this section, try the **string1.py** exercise in the Basic Exercises.

Python Lists

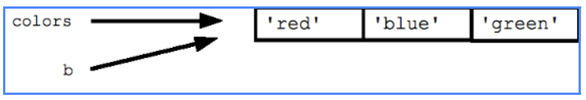
Python has a great built-in list type named "list". List literals are written within square brackets [ ]. Lists work similarly to strings -- use the len() function and square brackets [ ] to access data, with the first element at index 0. (See the official python.org list docs.)

colors 'red''blue''green'print colors## redprint colors## greenprintcolors



Assignment with an = on lists does not make a copy. Instead, assignment makes the two variables point to the one list in memory.

colors ## Does not copy the list



The "empty list" is just an empty pair of brackets [ ]. The '+' works to append two lists, so [1, 2] + [3, 4] yields [1, 2, 3, 4] (this is just like + with strings).

FOR and IN

Python's *for* and *in* constructs are extremely useful, and the first use of them we'll see is with lists. The *for* construct -- for **var** in **list** -- is an easy way to look at each element in a list (or other collection). Do not add or remove from the list during iteration.

squares squaresprint

If you know what sort of thing is in the list, use a variable name in the loop that captures that information such as "num", or "name", or "url". Since python code does not have other syntax to remind you of types, your variable names are a key way for you to keep straight what is going on.

The *in* construct on its own is an easy way to test if an element appears in a list (or other collection) -- **value** in **collection** -- tests if the value is in the collection, returning True/False.

'larry''curly''moe''curly'print'yay'

The for/in constructs are very commonly used in Python code and work on data types other than list, so should just memorize their syntax. You may have habits from other languages where you start manually iterating over a collection, where in Python you should just use for/in.

You can also use for/in to work on a string. The string acts like a list of its chars, so for ch in s: print ch prints all the chars in a string.

Range

The range(n) function yields the numbers 0, 1, ... n-1, and range(a, b) returns a, a+1, ... b-1 -- up to but not including the last number. The combination of the for-loop and the range() function allow you to build a traditional numeric for loop:

print the numbers from 0 through 99 rangeprint

There is a variant xrange() which avoids the cost of building the whole list for performance sensitive cases (in Python 3000, range() will have the good performance behavior and you can forget about xrange()).

While Loop

Python also has the standard while-loop, and the *break* and *continue* statements work as in C++ and Java, altering the course of the innermost loop. The above for/in loops solves the common case of iterating over every element in a list, but the while loop gives you total control over the index numbers. Here's a while loop which accesses every 3rd element in a list:

Access every 3rd element in a listwhileprint

List Methods

Here are some other common list methods.

- list.append(elem) -- adds a single element to the end of the list. Common error: does not return the new list, just modifies the original.
- list.insert(index, elem) -- inserts the element at the given index, shifting elements to the right.
- list.extend(list2) adds the elements in list2 to the end of the list. Using + or += on a list is similar to using extend().
- list.index(elem) -- searches for the given element from the start of the list and returns its index. Throws a ValueError if the element does not appear (use "in" to check without a ValueError).
- list.remove(elem) -- searches for the first instance of the given element and removes it (throws ValueError if not present)
- list.sort() -- sorts the list in place (does not return it). (The sorted() function shown below is preferred.)
- list.reverse() -- reverses the list in place (does not return it)
- list.pop(index) -- removes and returns the element at the given index. Returns the rightmost element if index is omitted (roughly the opposite of append()).

Notice that these are *methods* on a list object, while len() is a function that takes the list (or string or whatever) as an argument.

'larry''curly''moe'append'shemp'## append elem at endinsert'xxx'## insert elem at index 0extend'yyy''zzz'## add list of elems at endprint## ['xxx', 'larry', 'curly', 'moe', 'shemp', 'yyy', 'zzz']printindex'curly'remove'curly'## search and remove that element## removes and returns 'larry'print## ['xxx', 'moe', 'shemp', 'yyy', 'zzz']

Common error: note that the above methods do not *return* the modified list, they just modify the original list.

printappend## NO, does not work, append() returns None## Correct pattern:appendprint## [1, 2, 3, 4]

List Build Up

One common pattern is to start a list a the empty list [], then use append() or extend() to add elements to it:

Start as the empty listappend## Use append() to add elementsappend

List Slices

Slices work on lists just as with strings, and can also be used to change sub-parts of the list.

print## ['b', 'c']## replace ['a', 'b'] with ['z']print## ['z', 'c', 'd']

Exercise: list1.py

To practice the material in this section, try the problems in **list1.py** that do not use sorting (in the Basic Exercises).

Python Sorting

The easiest way to sort is with the sorted(list) function, which takes a list and returns a new list with those elements in sorted order. The original list is not changed.

print sorted## [1, 3, 4, 5]print## [5, 1, 4, 3]

It's most common to pass a list into the sorted() function, but in fact it can take as input any sort of iterable collection. The older list.sort() method is an alternative detailed below. The sorted() function seems easier to use compared to sort(), so I recommend using sorted().

The sorted() function can be customized though optional arguments. The sorted() optional argument reverse=True, e.g. sorted(list, reverse=True), makes it sort backwards.

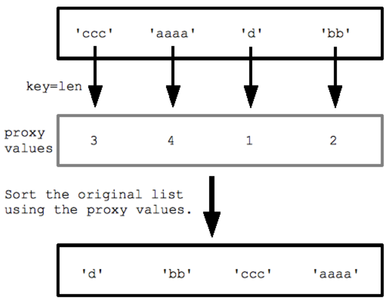
print sorted## ['BB', 'CC', 'aa', 'zz'] (case sensitive)print sorted reverse## ['zz', 'aa', 'CC', 'BB']

Custom Sorting With key=

For more complex custom sorting, sorted() takes an optional "key=" specifying a "key" function that transforms each element before comparison. The key function takes in 1 value and returns 1 value, and the returned "proxy" value is used for the comparisons within the sort.

For example with a list of strings, specifying key=len (the built in len() function) sorts the strings by length, from shortest to longest. The sort calls len() for each string to get the list of proxy length values, and the sorts with those proxy values.

'ccc''aaaa'print sorted## ['d', 'bb', 'ccc', 'aaaa']



As another example, specifying "str.lower" as the key function is a way to force the sorting to treat uppercase and lowercase the same:

"key" argument specifying str.lower function to use for sortingprint sortedlower## ['aa', 'BB', 'CC', 'zz']

You can also pass in your own MyFn as the key function, like this:

Say we have a list of strings we want to sort by the last letter of the string.## Write a little function that takes a string, and returns its last letter.## This will be the key function (takes in 1 value, returns 1 value).return## Now pass key=MyFn to sorted() to sort by the last letter:print sorted## ['wa', 'zb', 'xc', 'yd']

To use key= custom sorting, remember that you provide a function that takes one value and returns the proxy value to guide the sorting. There is also an optional argument "cmp=cmpFn" to sorted() that specifies a traditional two-argument comparison function that takes two values from the list and returns negative/0/positive to indicate their ordering. The built in comparison function for strings, ints, ... is cmp(a, b), so often you want to call cmp() in your custom comparator. The newer one argument key= sorting is generally preferable.

sort() method

As an alternative to sorted(), the sort() method on a list sorts that list into ascending order, e.g. list.sort(). The sort() method changes the underlying list and returns None, so use it like this:

```
 alist## correct

 alist  blist## NO incorrect, sort() returns None
```

The above is a very common misunderstanding with sort() -- it *does not return* the sorted list. The sort() method must be called on a list; it does not work on any enumerable collection (but the sorted() function above works on anything). The sort() method predates the sorted() function, so you will likely see it in older code. The sort() method does not need to create a new list, so it can be a little faster in the case that the elements to sort are already in a list.

Tuples

A tuple is a fixed size grouping of elements, such as an (x, y) co-ordinate. Tuples are like lists, except they are immutable and do not change size (tuples are not strictly immutable since one of the contained elements could be mutable). Tuples play a sort of "struct" role in Python -- a convenient way to pass around a little logical, fixed size bundle of values. A function that needs to return multiple values can just return a tuple of the values. For example, if I wanted to have a list of 3-d coordinates, the natural python representation would be a list of tuples, where each tuple is size 3 holding one (x, y, z) group.

To create a tuple, just list the values within parenthesis separated by commas. The "empty" tuple is just an empty pair of parenthesis. Accessing the elements in a tuple is just like a list -- len(), [ ], for, in, etc. all work the same.

```
 tuple printtupleprint tuple

 tuple'bye'## NO, tuples cannot be changed

 tuple 'bye'## this works
```

To create a size-1 tuple, the lone element must be followed by a comma.

```
 tuple ## size-1 tuple
```

It's a funny case in the syntax, but the comma is necessary to distinguish the tuple from the ordinary case of putting an expression in parentheses. In some cases you can omit the parenthesis and Python will see from the commas that you intend a tuple.

Assigning a tuple to an identically sized tuple of variable names assigns all the corresponding values. If the tuples are not the same size, it throws an error. This feature works for lists too.

"hike"print## hikeerr_string err_code## Foo() returns a length-2 tuple

List Comprehensions (optional)

List comprehensions are a more advanced feature which is nice for some cases but is not needed for the exercises and is not something you need to learn at first (i.e. you can skip this section). A list comprehension is a compact way to write an expression that expands to a whole list. Suppose we have a list nums [1, 2, 3], here is the list comprehension to compute a list of their squares [1, 4, 9]:

```
 squares ## [1, 4, 9, 16]
```

The syntax is [*expr* for var in list] -- the for var in list looks like a regular for-loop, but without the colon (:). The *expr* to its left is evaluated once for each element to give the values for the new list. Here is an example with strings, where each string is changed to upper case with '!!!' appended:

'hello''and''goodbye'

shouting upper'!!!'## ['HELLO!!!', 'AND!!!', 'GOODBYE!!!']

You can add an if test to the right of the for-loop to narrow the result. The if test is evaluated for each element, including only the elements where the test is true.

Select values <= 2

```
 small <=## [2, 1]## Select fruits containing 'a', change to upper case

 fruits 'apple''cherry''bannana''lemon'

 afruits upper fruits ## ['APPLE', 'BANNANA']
```

Exercise: list1.py

To practice the material in this section, try later problems in **list1.py** that use sorting and tuples (in the Basic Exercises).

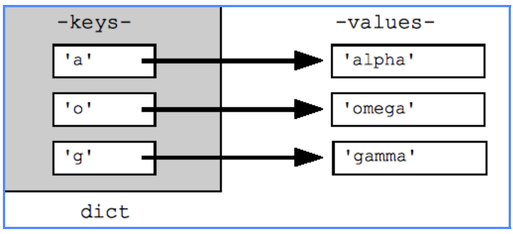
Python Dict and File

Dict Hash Table

Python's efficient key/value hash table structure is called a "dict". The contents of a dict can be written as a series of key:value pairs within braces { }, e.g. dict = {key1:value1, key2:value2, ... }. The "empty dict" is just an empty pair of curly braces {}.

Looking up or setting a value in a dict uses square brackets, e.g. dict['foo'] looks up the value under the key 'foo'. Strings, numbers, and tuples work as keys, and any type can be a value. Other types may or may not work correctly as keys (strings and tuples work cleanly since they are immutable). Looking up a value which is not in the dict throws a KeyError -- use "in" to check if the key is in the dict, or use dict.get(key) which returns the value or None if the key is not present (or get(key, not-found) allows you to specify what value to return in the not-found case).

Can build up a dict by starting with the the empty dict {}## and storing key/value pairs into the dict like this:## dict[key] = value-for-that-key'alpha''gamma''omega'print## {'a': 'alpha', 'o': 'omega', 'g': 'gamma'}print## Simple lookup, returns 'alpha'## Put new key/value into dict## True## print dict['z'] ## Throws KeyErrorprint## Avoid KeyErrorprint## None (instead of KeyError)



A for loop on a dictionary iterates over its keys by default. The keys will appear in an arbitrary order. The methods dict.keys() and dict.values() return lists of the keys or values explicitly. There's also an items() which returns a list of (key, value) tuples, which is the most efficient way to examine all the key value data in the dictionary. All of these lists can be passed to the sorted() function.

By default, iterating over a dict iterates over its keys.## Note that the keys are in a random order.print## prints a g o## Exactly the same as aboveprint## Get the .keys() list:print## ['a', 'o', 'g']## Likewise, there's a .values() list of valuesprintvalues## ['alpha', 'omega', 'gamma']## Common case -- loop over the keys in sorted order,## accessing each key/value sortedprint## .items() is the dict expressed as (key, value) tuplesprintitems## [('a', 'alpha'), ('o', 'omega'), ('g', 'gamma')]## This loop syntax accesses the whole dict by looping## over the .items() tuple list, accessing one (key, value)## pair on each iteration.itemsprint'>'## a > alpha o > omega g > gamma

There are "iter" variants of these methods called iterkeys(), itervalues() and iteritems() which avoid the cost of constructing the whole list -- a performance win if the data is huge. However, I generally prefer the plain keys() and values() methods with their sensible names. In Python 3000 revision, the need for the iterkeys() variants is going away.

Strategy note: from a performance point of view, the dictionary is one of your greatest tools, and you should use where you can as an easy way to organize data. For example, you might read a log file where each line begins with an ip address, and store the data into a dict using the ip address as the key, and the list of lines where it appears as the value. Once you've read in the whole file, you can look up any ip address and instantly see its list of lines. The dictionary takes in scattered data and make it into something coherent.

Dict Formatting

The % operator works conveniently to substitute values from a dict into a string by name:

'word''garfield''count''I want %(count)d copies of %(word)s'# %d for int, %s for string# 'I want 42 copies of garfield'

The "del" operator does deletions. In the simplest case, it can remove the definition of a variable, as if that variable had not been defined. Del can also be used on list elements or slices to delete that part of the list and to delete entries from a dictionary.

var no more!## Delete first element## Delete last two elementsprint## ['b']## Delete 'b' entryprint## {'a':1, 'c':3}

Files

The open() function opens and returns a file handle that can be used to read or write a file in the usual way. The code f = open('name', 'r') opens the file into the variable f, ready for reading operations, and use f.close() when finished. Instead of 'r', use 'w' for writing, and 'a' for append. The special mode 'rU' is the "Universal" option for text files where it's smart about converting different line-endings so they always come through as a simple '\n'. The standard for-loop works for text files, iterating through the lines of the file (this works only for text files, not binary files). The for-loop technique is a simple and efficient way to look at all the lines in a text file:

Echo the contents of a file'foo.txt'## iterates over the lines of the fileprint## trailing , so print does not add an end-of-line char## since 'line' already includes the end-of line.close

Reading one line at a time has the nice quality that not all the file needs to fit in memory at one time -- handy if you want to look at every line in a 10 gigabyte file without using 10 gigabytes of memory. The f.readlines() method reads the whole file into memory and returns its contents as a list of its lines. The f.read() method reads the whole file into a single string, which can be a handy way to deal with the text all at once, such as with regular expressions we'll see later.

For writing, f.write(string) method is the easiest way to write data to an open output file. Or you can use "print" with an open file, but the syntax is nasty: "print >> f, string". In python 3000, the print syntax will be fixed to be a regular function call with a file= optional argument: "print(string, file=f)".

Files Unicode

The "codecs" module provides support for reading a unicode file.

import codecs

f codecs'foo.txt''utf-8'# here line is a *unicode* string

For writing, use f.write() since print does not fully support unicode.

Exercise Incremental Development

Building a Python program, don't write the whole thing in one step. Instead identify just a first milestone, e.g. "well the first step is to extract the list of words." Write the code to get to that milestone, and just print your data structures at that point, and then you can do a sys.exit(0) so the program does not run ahead into its not-done parts. Once the milestone code is working, you can work on code for the next milestone. Being able to look at the printout of your variables at one state can help you think about how you need to transform those variables to get to the next state. Python is very quick with this pattern, allowing you to make a little change and run the program to see how it works. Take advantage of that quick turnaround to build your program in little steps.

Exercise: wordcount.py

Combining all the basic Python material -- strings, lists, dicts, tuples, files -- try the summary **wordcount.py** exercise in the Basic Exercises.

Python Regular Expressions

Regular expressions are a powerful language for matching text patterns. This page gives a basic introduction to regular expressions themselves sufficient for our Python exercises and shows how regular expressions work in Python. The Python "re" module provides regular expression support.

In Python a regular expression search is typically written as:

```
match search
```

The re.search() method takes a regular expression pattern and a string and searches for that pattern within the string. If the search is successful, search() returns a match object or None otherwise. Therefore, the search is usually immediately followed by an if-statement to test if the search succeeded, as shown in the following example which searches for the pattern 'word:' followed by a 3 letter word (details below):

```
'an example word:cat!!'

match search'word:\w\w\w'# If-statement after search() tests if it succeeded matchprint'found' matchgroup## 'found word:cat'print'did not find'
```

The code match = re.search(pat, str) stores the search result in a variable named "match". Then the if-statement tests the match -- if true the search succeeded and match.group() is the matching text (e.g. 'word:cat'). Otherwise if the match is false (None to be more specific), then the search did not succeed, and there is no matching text.

The 'r' at the start of the pattern string designates a python "raw" string which passes through backslashes without change which is very handy for regular expressions (Java needs this feature badly!). I recommend that you always write pattern strings with the 'r' just as a habit.

Basic Patterns

The power of regular expressions is that they can specify patterns, not just fixed characters. Here are the most basic patterns which match single chars:

- a, X, 9, < -- ordinary characters just match themselves exactly. The meta-characters which do not match themselves because they have special meanings are: . ^ $ * + ? { [ ] \ | ( ) (details below)
- . (a period) -- matches any single character except newline '\n'
- \w -- (lowercase w) matches a "word" character: a letter or digit or underbar [a-zA-Z0-9_]. Note that although "word" is the mnemonic for this, it only matches a single word char, not a whole word. \W (upper case W) matches any non-word character.
- \b -- boundary between word and non-word
- \s -- (lowercase s) matches a single whitespace character -- space, newline, return, tab, form [\n\r\t\f]. \S (upper case S) matches any non-whitespace character.
- \t, \n, \r -- tab, newline, return
- \d -- decimal digit [0-9] (some older regex utilities do not support but \d, but they all support \w and \s)
- ^ = start, $ = end -- match the start or end of the string
- \ -- inhibit the "specialness" of a character. So, for example, use \. to match a period or \\ to match a slash. If you are unsure if a character has special meaning, such as '@', you can put a slash in front of it, \@, to make sure it is treated just as a character.

Basic Examples

Joke: what do you call a pig with three eyes? piiig!

The basic rules of regular expression search for a pattern within a string are:

- The search proceeds through the string from start to end, stopping at the first match found
- All of the pattern must be matched, but not all of the string
- If match = re.search(pat, str) is successful, match is not None and in particular match.group() is the matching text

Search for pattern 'iii' in string 'piiig'.## All of the pattern must match, but it may appear anywhere.## On success, match.group() is matched text.

```
match search'iii''piiig'=>  found matchgroup*iii*

match search'igs''piiig'=> found match ## . = any char but \n

match search'..g''piiig'=>  found matchgroup*iig'## \d = digit char, \w = word char

match search'\d\d\d''p123g'=>  found matchgroup*123*

match search'\w\w\w''@@abc!!'=>  found matchgroup*abc*
```

Repetition

Things get more interesting when you use + and * to specify repetition in the pattern

- + -- 1 or more occurrences of the pattern to its left, e.g. 'i+' = one or more i's
- * -- 0 or more occurrences of the pattern to its left
- ? -- match 0 or 1 occurrences of the pattern to its left

Leftmost & Largest

First the search finds the leftmost match for the pattern, and second it tries to use up as much of the string as possible -- i.e. + and * go as far as possible (the + and * are said to be "greedy").

Repetition Examples

i+ = one or more i's, as many as possible.

```
match search'pi+''piiig'=>  found matchgroup*piii'## Finds the first/leftmost solution, and within it drives the +## as far as possible (aka 'leftmost and largest').## In this example, note that it does not get to the second set of i's.

match search'piigiiii'=>  found matchgroup## \s* = zero or more whitespace chars## Here look for 3 digits, possibly separated by whitespace.

match search'\d\s*\d\s*\d''xx1 2  3xx'=>  found matchgroup*1 2  3'

match search'\d\s*\d\s*\d''xx12  3xx'=>  found matchgroup*12  3'

match search'\d\s*\d\s*\d''xx123xx'=>  found matchgroup*123'## ^ = matches the start of string, so this fails:

match search'^b\w+''foobar'=> found match ## but without the ^ it succeeds:

match search'b\w+''foobar'=>  found matchgroup*bar'
```

Emails Example

Suppose you want to find the email address inside the string 'xyz alice-b@google.com purple monkey'. We'll use this as a running example to demonstrate more regular expression features. Here's an attempt using the pattern r'\w+@\w+':

```
'purple alice-b@google.com monkey dishwasher'

match search'\w+@\w+' matchprint matchgroup## 'b@google'
```

The search does not get the whole email address in this case because the \w does not match the '-' or '.' in the address. We'll fix this using the regular expression features below.

Square Brackets

Square brackets can be used to indicate a set of chars, so [abc] matches 'a' or 'b' or 'c'. The codes \w, \s etc. work inside square brackets too with the one exception that dot (.) just means a literal dot. For the emails problem, the square brackets are an easy way to add '.' and '-' to the set of chars which can appear around the @ with the pattern r'[\w.-]+@[\w.-]+' to get the whole email address:

```
match search'[\w.-]+@[\w.-]+' matchprint matchgroup## 'alice-b@google.com'
```

(More square-bracket features) You can also use a dash to indicate a range, so [a-z] matches all lowercase letters. To use a dash without indicating a range, put the dash last, e.g. [abc-]. An up-hat (^) at the start of a square-bracket set inverts it, so [^ab] means any char except 'a' or 'b'.

Group Extraction

The "group" feature of a regular expression allows you to pick out parts of the matching text. Suppose for the emails problem that we want to extract the username and host separately. To do this, add parenthesis ( ) around the username and host in the pattern, like this: r'([\w.-]+)@([\w.-]+)'. In this case, the parenthesis do not change what the pattern will match, instead they establish logical "groups" inside of the match text. On a successful search, match.group(1) is the match text corresponding to the 1st left parenthesis, and match.group(2) is the text corresponding to the 2nd left parenthesis. The plain match.group() is still the whole match text as usual.

```
'purple alice-b@google.com monkey dishwasher'

match search'([\w.-]+)@([\w.-]+)' matchprint matchgroup## 'alice-b@google.com' (the whole match)print matchgroup## 'alice-b' (the username, group 1)print matchgroup## 'google.com' (the host, group 2)
```

A common workflow with regular expressions is that you write a pattern for the thing you are looking for, adding parenthesis groups to extract the parts you want.

findall

findall() is probably the single most powerful function in the re module. Above we used re.search() to find the first match for a pattern. findall() finds *all* the matches and returns them as a list of strings, with each string representing one match.

```
## Suppose we have a text with many email addresses'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'## Here re.findall() returns a list of all the found email strings

 emails findall'[\w.-]+@[\w.-]+'## ['alice@google.com', 'bob@abc.com'] email  emails## do something with each found email stringprint email
```

findall With Files

For files, you may be in the habit of writing a loop to iterate over the lines of the file, and you could then call findall() on each line. Instead, let findall() do the iteration for you -- much better! Just feed the whole file text into findall() and let it return a list of all the matches in a single step (recall that f.read() returns the whole text of a file in a single string):

```
# Open file'test.txt'# Feed the file text into findall(); it returns a list of all the found strings

 strings findall'some pattern'
```

findall and Groups

The parenthesis ( ) group mechanism can be combined with findall(). If the pattern includes 2 or more parenthesis groups, then instead of returning a list of strings, findall() returns a list of *tuples*. Each tuple represents one match of the pattern, and inside the tuple is the group(1), group(2) .. data. So if 2 parenthesis groups are added to the email pattern, then findall() returns a list of tuples, each length 2 containing the username and host, e.g. ('alice', 'google.com').

```
'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'

 tuples findall'([\w.-]+)@([\w.-]+)'print tuples  ## [('alice', 'google.com'), ('bob', 'abc.com')] tuple  tuplesprint tuple## username‌print tuple## host
```

Once you have the list of tuples, you can loop over it to do some computation for each tuple. If the pattern includes no parenthesis, then findall() returns a list of found strings as in earlier examples. If the pattern includes a single set of parenthesis, then findall() returns a list of strings corresponding to that single group. (Obscure optional feature: Sometimes you have paren ( ) groupings in the pattern, but which you do not want to extract. In that case, write the parens with a ?: at the start, e.g. (?:) and that left paren will not count as a group result.)

RE Workflow and Debug

Regular expression patterns pack a lot of meaning into just a few characters , but they are so dense, you can spend a lot of time debugging your patterns. Set up your runtime so you can run a pattern and print what it matches easily, for example by running it on a small test text and printing the result of findall(). If the pattern matches nothing, try weakening the pattern, removing parts of it so you get too many matches. When it's matching nothing, you can't make any progress since there's nothing concrete to look at. Once it's matching too much, then you can work on tightening it up incrementally to hit just what you want.

Options

The re functions take options to modify the behavior of the pattern match. The option flag is added as an extra argument to the search() or findall() etc., e.g. re.search(pat, str, re.IGNORECASE).

- IGNORECASE -- ignore upper/lowercase differences for matching, so 'a' matches both 'a' and 'A'.
- DOTALL -- allow dot (.) to match newline -- normally it matches anything but newline. This can trip you up -- you think .* matches everything, but by default it does not go past the end of a line. Note that \s (whitespace) includes newlines, so if you want to match a run of whitespace that may include a newline, you can just use \s*
- MULTILINE -- Within a string made of many lines, allow ^ and $ to match the start and end of each line. Normally ^/$ would just match the start and end of the whole string.

Greedy vs. Non-Greedy (optional)

This is optional section which shows a more advanced regular expression technique not needed for the exercises.

Suppose you have text with tags in it: foo and so on

Suppose you are trying to match each tag with the pattern '(<.*>)' -- what does it match first?

The result is a little surprising, but the greedy aspect of the .* causes it to match the whole 'foo and so on' as one big match. The problem is that the .* goes as far as is it can, instead of stopping at the first > (aka it is "greedy").

There is an extension to regular expression where you add a ? at the end, such as .*? or .+?, changing them to be non-greedy. Now they stop as soon as they can. So the pattern '(<.*?>)' will get just '' as the first match, and '' as the second match, and so on getting each <..> pair in turn. The style is typically that you use a .*?, and then immediately its right look for some concrete marker (> in this case) that forces the end of the .*? run.

The '?' extension originated in Perl, and regular expressions that include Perl's extensions are known as Perl Compatible Regular Expressions -- pcre. Python includes pcre support. Many command line utils etc. have a flag where they accept pcre patterns.

An older but widely used technique to code this idea of "all of these chars except stopping at X" uses the square-bracket style. For the above you could write the pattern, but instead of .* to get all the chars, use [^>]* which skips over all characters which are not > (the leading ^ "inverts" the square bracket set, so it matches any char not in the brackets).

Substitution (optional)

The re.sub(pat, replacement, str) function searches for all the instances of pattern in the given string, and replaces them. The replacement string can include '\1', '\2' which refer to the text from group(1), group(2), and so on from the original matching text.

Here's an example which searches for all the email addresses, and changes them to keep the user (\1) but have yo-yo-dyne.com as the host.

```
'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'## re.sub(pat, replacement, str) -- returns new string with all replacements,## \1 is group(1), \2 group(2) in the replacementprint'([\w.-]+)@([\w.-]+)'r'\1@yo-yo-dyne.com'## purple alice@yo-yo-dyne.com, blah monkey bob@yo-yo-dyne.com blah dishwasher
```

Exercise

To practice regular expressions, see the Baby Names Exercise.

Python Utilities

In this section, we look at a few of Python's many standard utility modules to solve common problems.

File System -- os, os.path, shutil

The *os* and *os.path* modules include many functions to interact with the file system. The *shutil* module can copy files.

- os module docs
- filenames = os.listdir(dir) -- list of filenames in that directory path (not including . and ..). The filenames are just the names in the directory, not their absolute paths.
- os.path.join(dir, filename) -- given a filename from the above list, use this to put the dir and filename together to make a path
- os.path.abspath(path) -- given a path, return an absolute form, e.g. /home/nick/foo/bar.html
- os.path.dirname(path), os.path.basename(path) -- given dir/foo/bar.html, return the dirname "dir/foo" and basename "bar.html"
- os.path.exists(path) -- true if it exists
- os.mkdir(dir_path) -- makes one dir, os.makedirs(dir_path) makes all the needed dirs in this path
- shutil.copy(source-path, dest-path) -- copy a file (dest path directories should exist)

Example pulls filenames from a dir, prints their relative and absolute paths printdir

filenames listdir filename filenamesprint filename ## foo.txtprint filename## dir/foo.txt (relative to current dir)printabspath filename## /home/nick/dir/foo.txt

Exploring a module works well with the built-in python help() and dir() functions. In the interpreter, do an "import os", and then use these commands look at what's available in the module: dir(os), help(os.listdir), dir(os.path), help(os.path.dirname).

Running External Processes -- commands

The *commands* module is a simple way to run an external command and capture its output.

- commands module docs
- (status, output) = commands.getstatusoutput(cmd) -- runs the command, waits for it to exit, and returns its status int and output text as a tuple. The command is run with its standard output and standard error combined into the one output text. The status will be non-zero if the command failed. Since the standard-err of the command is captured, if it fails, we need to print some indication of what happened.
- output = commands.getoutput(cmd) -- as above, but without the status int.
- There is a commands.getstatus() but it does something else, so don't use it -- dumbest bit of method naming ever!
- If you want more control over the running of the sub-process, see the "popen2" module (http://docs.python.org/lib/module-popen2.html)
- There is also a simple os.system(cmd) which runs the command and dumps its output onto your output and returns its error code. This works if you want to run the command but do not need to capture its output into your python data structures.

Given a dir path, run an external 'ls -l' on it --## shows how to call an external program listdir'ls -l 'print"Command to run:"## good to debug cmd before actually running itstatus output commandsgetstatusoutput status## Error case, print the command's output to stderr and exitstderrwriteoutputprint output ## Otherwise do something with the command's output

Exceptions

An exception represents a run-time error that halts the normal execution at a particular line and transfers control to error handling code. This section just introduces the most basic uses of exceptions. For example a run-time error might be that a variable used in the program does not have a value (ValueError .. you've probably seen that one a few times), or a file open operation error because that a does not exist (IOError). (See [[http://docs.python.org/tut/node10.html][exception docs]])

Without any error handling code (as we have done thus far), a run-time exception just halts the program with an error message. That's a good default behavior, and you've seen it many times. You can add a "try/except" structure to your code to handle exceptions, like this:

Either of these two lines could throw an IOError, say## if the file does not exist or the read() encounters a low level error.filenamecloseexceptIOError## Control jumps directly to here if any of the above lines throws IOError.stderrwrite'problem reading:' filename## In any case, the code then continues with the line after the try/except

The try: section includes the code which might throw an exception. The except: section holds the code to run if there is an exception. If there is no exception, the except: section is skipped (that is, that code is for error handling only, not the "normal" case for the code). You can get a pointer to the exception object itself with syntax "except IOError, e: .. (e points to the exception object)".

HTTP -- urllib and urlparse

The module *urllib* provides url fetching -- making a url look like a file you can read form. The *urlparse* module can take apart and put together urls.

- urllib module docs
- ufile = urllib.urlopen(url) -- returns a file like object for that url
- text = ufile.read() -- can read from it, like a file (readlines() etc. also work)
- info = ufile.info() -- the meta info for that request. info.gettype() is the mime time, e.g. 'text/html'
- baseurl = ufile.geturl() -- gets the "base" url for the request, which may be different from the original because of redirects
- urllib.urlretrieve(url, filename) -- downloads the url data to the given file path
- urlparse.urljoin(baseurl, url) -- given a url that may or may not be full, and the baseurl of the page it comes from, return a full url. Use geturl() above to provide the base url.

Given a url, try to retrieve it. If it's text/html,## print its base url and its text.

ufile urlliburlopen## get file-like object for url ufile## meta-info about the url contentgettype'text/html'print'base url:' ufilegeturl ufile## read all its textprint

The above code works fine, but does not include error handling if a url does not work for some reason. Here's a version of the function which adds try/except logic to print an error message if the url operation fails.

Version that uses try/except to print an error message if the## urlopen() fails. wget2

ufile urlliburlopen ufilegettype'text/html'print ufileexceptIOErrorprint'problem reading url:'

Exercise

To practice the file system and external-commands material, see the Copy Special Exercise. To practice the urllib material, see the Log Puzzle Exercise.