Python

- 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.

- 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 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".

- 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.

User-defined Functions

```
# Defines a "repeat" function that takes 2 arguments.
def repeat(s, exclaim):
  """
  Returns the string 's' repeated 3 times.
  If exclaim is true, add exclamation marks.
  """
  result = s + s + s # can also use "s * 3" which is faster (Why?)
  if exclaim
    result = result + '!!!'
  return result
```

- 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.
- 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.

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.

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.

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 span 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'.
- 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..
- 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.
- https://docs.python.org/3/library/stdtypes.html#string-methods

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 parentheses):

```
# % operator
text = "%d little pigs come out or I'll %s and %s and %s" % (3, 'huff', 'pu
ff', 'blow down')
```

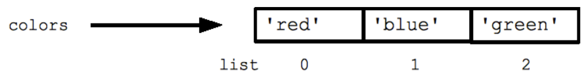
```
 # add parens to make the long-line work:
 text = ("%d little pigs come out or I'll %s and %s and %s" %(3, 'huff', '
 puff', 'blow down')
```

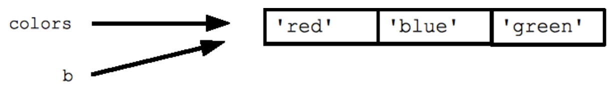
```
## (ustring from above contains a unicode string)
> s = ustring.encode('utf-8')
> s
'A unicode \xc6\x8e string \xc3\xb1'  ## bytes of utf-8 encoding
> t = unicode(s, 'utf-8')             ## Convert bytes back to a unicode st
ring
> t == ustring                      ## It's the same as the original, yay!
True
```

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.

```
  colors = ['red', 'blue', 'green']
  print colors[0]    ## red
  print colors[2]    ## green
  print len(colors)  ## 3
```





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).

```
squares = [1, 4, 9, 16]
 sum = 0
for num in squares:
 sum += num
print sum  ## 30
```

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.

```
list = ['larry', 'curly', 'moe']
if 'curly' in list:
 print 'yay'
```

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
for i in range(100):
print i
```

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.

```
  ## Access every 3rd element in a list
  i = 0
  while i < len(a):
    print a[i]
    i = i + 3
```

Python Sorting

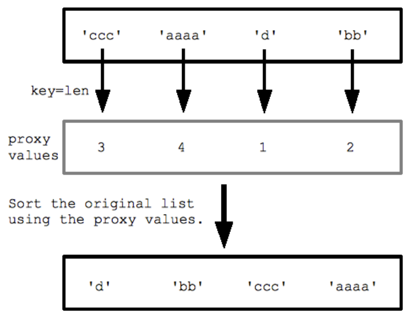
The easiest way to sort is with the sorted(list) function, which takes a list and returns a new list with those elements in sorted order. The original list is not changed.

```
a = [5, 1, 4, 3]
  print sorted(a)  ## [1, 3, 4, 5]
  print a  ## [5, 1, 4, 3]
```

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.

```
strs = ['ccc', 'aaaa', 'd', 'bb']
print sorted(strs, key=len)  ## ['d', 'bb', 'ccc', 'aaaa']
```



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).

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]:

```
nums = [1, 2, 3, 4]
squares = [ n * n for n in nums ]   ## [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:

```
  strs = ['hello', 'and', 'goodbye']
  shouting = [ s.upper() + '!!!' for s in strs ]
  ## ['HELLO!!!', 'AND!!!', 'GOODBYE!!!']
```

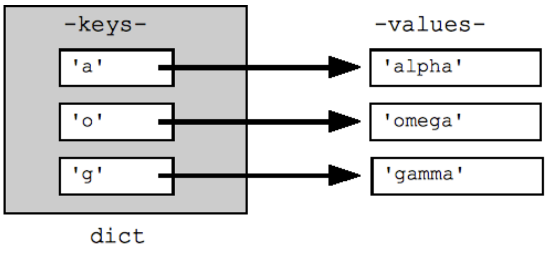
Dict Hash Table

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 {}.

```
 ## 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
 dict = {}
 dict['a'] = 'alpha'
 dict['g'] = 'gamma'
 dict['o'] = 'omega'

 print dict  ## {'a': 'alpha', 'o': 'omega', 'g': 'gamma'}
 print dict['a']     ## Simple lookup, returns 'alpha'
 dict['a'] = 6       ## Put new key/value into dict
 'a' in dict         ## True

 ## print dict['z']                ## Throws KeyError
if 'z' in dict: print dict['z']     ## Avoid KeyError
print dict.get('z')  ## None (instead of KeyError)
```



```
## By default, iterating over a dict iterates over its keys.
## Note that the keys are in a random order.
 for key in dict: print ke
 ## prints a g o
 ## Exactly the same as above

 for key in dict.keys(): print key

 ## Get the .keys() list:

 print dict.keys()  ## ['a', 'o', 'g']
 ## Likewise, there's a .values() list of values
 print dict.values()  ## ['alpha', 'omega', 'gamma']

 ## Common case -- loop over the keys in sorted order,
 ## accessing each key/value

 for key in sorted(dict.keys())
   print key, dict[key]

 ## .items() is the dict expressed as (key, value) tuples
```

```
  print dict.items()  ##  [('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.

  for k, v in dict.items(): print k, '>', v
## a > alpha    o > omega     g > gamma
```

Del

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 = 6
 del var  # var no more!


 list = ['a', 'b', 'c', 'd']
 del list[0]     ## Delete first element
 del list[-2:]   ## Delete last two elements
 print list      ## ['b']

 dict = {'a':1, 'b':2, 'c':3}
 del dict['b']   ## Delete 'b' entry
 print dict      ## {'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
  f = open('foo.txt', 'rU')
  for line in f:   ## iterates over the lines of the file
    print line,    ## trailing , so print does not add an end-of-line char
                   ## since 'line' already includes the end-of line
  f.close()
```

Files Unicode

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

```
import codecs
f = codecs.open('foo.txt', 'rU', 'utf-8')
for line in f:
  # here line is a *unicode* string
```

Regular Expressions

Regular expressions are a powerful language for matching text patterns.

```
str = 'an example word:cat!!'
match = re.search(r'word:\w\w\w', str)

# If-statement after search() tests if it succeeded
  if match:
    print 'found', match.group() ## 'found word:cat'
  else:
    print 'did not find'
```

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:

```
## 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 = re.search(r'iii', 'piiig') =>  found, match.group() == "iii"
match = re.search(r'igs', 'piiig') =>  not found, match == None

## . = any char but \n
match = re.search(r'..g', 'piiig') =>  found, match.group() == "iig"

## \d = digit char, \w = word char
match = re.search(r'\d\d\d', 'p123g') =>  found, match.group() == "123"
match = re.search(r'\w\w\w', '@@abcd!!') =>  found, match.group() == “abc"
```

```
## i+ = one or more i's, as many as possible
match = re.search(r'pi+', 'piiig') =>  found, match.group() == "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 = re.search(r'i+', 'piigiiii') =>  found, match.group() == "ii"
```

```
## \s* = zero or more whitespace chars
## Here look for 3 digits, possibly separated by whitespace.

match = re.search(r'\d\s*\d\s*\d', 'xx1 2   3xx') =>  found, match.group() == "1 2   3"
match = re.search(r'\d\s*\d\s*\d', 'xx12  3xx') =>  found, match.group() == "12  3"
match = re.search(r'\d\s*\d\s*\d', 'xx123xx') =>  found, match.group() == "123"

## ^ = matches the start of string, so this fails:
match = re.search(r'^b\w+', 'foobar') =>  not found, match == None

## but without the ^ it succeeds:
match = re.search(r'b\w+', 'foobar') =>  found, match.group() == “bar"
```

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 p
aths

def printdir(dir):
  filenames = os.listdir(dir)
  for filename in filenames:
    print filename  ## foo.txt
    print os.path.join(dir, filename) ## dir/foo.txt (relative to current d
ir)
    print os.path.abspath(os.path.join(dir, filename)) ## /home/nick/dir/fo
o.txt
```

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

def listdir(dir):
  cmd = 'ls -l ' + dir
  print "Command to run:", cmd   ## good to debug cmd before actually runni
ng it
  (status, output) = commands.getstatusoutput(cmd)
  if status:    ## Error case, print the command's output to stderr and exi
t
    sys.stderr.write(output)
    sys.exit(1)
  print 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:

```
 try:
   ## Either of these two lines could throw an IOError, say
   ## if the file does not exist or the read() encounters a low level error.

   f = open(filename, 'rU')
   text = f.read()
   f.close()
```

```
  except IOError:
    ## Control jumps directly to here if any of the above lines throws IOError.
    sys.stderr.write('problem reading:' + filename)
  ## In any case, the code then continues with the line after the try/except
```

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.

def wget(url):
  ufile = urllib.urlopen(url)  ## get file-like object for url
  info = ufile.info()   ## meta-info about the url content
  if info.gettype() == 'text/html':
    print 'base url:' + ufile.geturl()
    text = ufile.read()  ## read all its text
    print text
```

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.

def wget2(url):
  try:
    ufile = urllib.urlopen(url)
    if ufile.info().gettype() == 'text/html':
      print ufile.read()

  except IOError:
    print 'problem reading url:', url
```