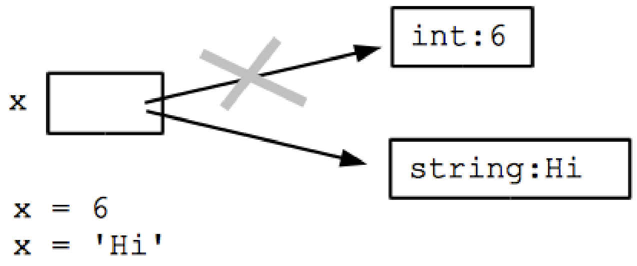
Python in One Easy Lesson

Nick Parlante Nov 2010

This is a one-hour introduction to Python used for Stanford's CS107. This material should work as an introduction for any experienced programmer. We'll look at some core Python features and get a feel for how it compares to other languages. For a longer introduction to Python by the same author with videos and exercises, see Google's Python Class. For online practice problems, see codingbat.com

Python is a popular open source, cross-platform language in the "dynamic language" niche, like Javascript, Ruby, Lisp, and Perl. See python.org home for docs, code, tutorials, etc.

Unlike C/C++, Python defers almost everything until runtime. Variables and functions do not have compile time types. Values at runtime are tagged with their type: int, string, ... Variables just point to value at runtime, and each value's type is used at runtime to resolve what the run should do. For example here's a variable x pointing first to an int, then to a string. The variable x does not have a type, but whatever value it points to does have a type.



This is how Python works, in contrast to the C/C++ style of knowing the type of every variable and using that for compile time code generation. Python does not generate code in that way at compile time. For example, consider this code fragment:

```
  x = x + x
```

In Python, x could be an int, or a string, or who knows what. Without that type information, code generation cannot work the way it does in C/C++. Instead, the Python interpreter looks at the types present in each bit of code in the moment at run time. This makes Python slower than C/C++, but also very flexible, and in any case it is missing both the advantages and disadvantages of a compile time type system. A Just In Time compiler (JIT) plays on this boundary -- working at run time, on the fly, to compile native code for code fragments. Since it's run time, the JIT can know the types stored in certain variables, etc. Java JIT technology is very mature. Python JITs are just getting started (see PyPy project).

Interpreter and Variables

You can run the Python interpreter and type code directly in to it -- a good way to try little experiments. The interpreter runs a read-eval-print loop with whatever you type. Don't need to declare variables and they do not have a fixed type. Reading from a variable that does not have a value is a runtime error. Python has a garbage collector (GC) provides automatic deallocation of unused memory.

```
$ python   ## start the interpreter
```

```
Python 2.4.4 (#1, Oct 18 2006, 10:34:39)
[GCC 4.0.1 (Apple Computer, Inc. build 5341)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>>
>>> a = 'hello'
>>> a      ## evaluate a to see its value
'hello'
>>> a = 6  ## assign 6 to a (GC takes care of 'hello')
>>> a
6
>>> b
Traceback (most recent call last):
  File "", line 1, in ?
NameError: name 'b' is not defined
```

Strings

Strings are delimited with ' or " or """, and are immutable. The len() function used to get length of all sorts of python types. Call len(s) to get length of string and s[0] accesses individual chars. There is no separate char type -- just a string length 1.
See the many string methods at: http://docs.python.org/library/stdtypes.html#string-methods

```
>>> a = 'hello'
>>> len(a)
5
>>> a[0]
'h'
>>> a + '!!!'  ## + works
'hello!!!'
>>> a + str(4)  ## must convert int->str to use with +
'hello4'
>>> a.upper()  ## .upper()/.lower() methods, original unchanged
'HELLO'
>>> a
'hello'
>>> a.isalpha()  ## True if all chars are alphabetic
True
```

Lists [ ], For Loop

List literals are enclosed in square brackets. Lists can contain any type of value. Use the len() function and [] to access elements (same as string). The .append() method appends an element to a list. See the official docs for the many available list methods:
http://docs.python.org/tutorial/datastructures.html
Important foreach syntax on lists: for **var** in **list**:

```
>>> a = ['hello', 'and', 'goodbye']
>>> len(a)
3
>>> a[0]
'hello'
>>> for word in a:
      print word
hello
and
goodbye
```

```
>>> 'goodbye' in a  ## use 'in' by itself to test containment
True
## .append() appends to the end
>>> a = []   # empty
>>> a.append(1)
>>> a.append(2)
>>> a
[1, 2]
## "slice" syntax a[1:] is sublist starting at index 1
>>> a[1:]
['and', 'goodbye']
## slice stops before index 2
>>> a[:2]
['hello', 'and']
## .split() on a string splits on whitespace to return a list
>>> 'hello and goodbye'.split()
['hello', 'and', 'goodbye']
```

Program syntax, if/else, Indentation

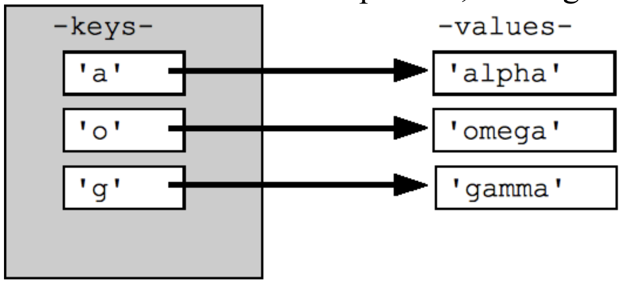
Below is the complete text of a **cat.py** file which prints out text file contents.

```
#!/usr/bin/python -tt

"""
cat.py -- just a little example python program
showing some common syntax. This program works.
nick.parlante@cs.stanford.edu

-tt flag above detects space/tab indent problems
"""

# sys is one of many available modules of library code, import to use.
# sys.argv is the list of command line arguments.
import sys

# defines a global variable
a = 123

# defines a 'cat' function which takes a filename
def cat(filename):
  """Given filename, print its text contents."""
  print filename, '======='
  f = open(filename, 'r')
  for line in f:  # goes through a text file line by line
    print line,   # trailing comma inhibits the ending print-newline
  # alternative, read the whole file into a single string:
  # text = f.read()
  f.close()

def main():
  # sys.argv contains command line arguments.
  # This assigns a list of all but the first arg into a local 'args' var.
  args = sys.argv[1:]

  # important syntax -- loop of variable 'filename' over the args list.
  for filename in args:
    # detect scary filenames: if/else and/or/not
    if filename == 'voldemort' or filename == 'vader':
      print 'this file is very worrying'
      cat(filemane, 123, bad_variable)
      # important point: errors in above line only caught if it is run
    else:
      # regular case
      cat(filename)
  print 'all done'  # this print is outside the loop, due to its indentation


# Standard boilerplate at end of file to call main() function.
if __name__ == '__main__':
  main()
```

The indention of each line controls whether it is within a loop, if statement, etc. -- there are no { } to define blocks of code. This use of indentation in Python is unusual, but it's logical and you get used to it. It's best to not use tab characters within your Python code; just use spaces.

Dict { } Hash Table

The built-in dict (i.e. hash-table) type uses { } for literals. Use dict[key-val] both to read and set entries in the dict. Build up a dict, starting it as {}, then assigning in the desired values.



```
>>> dict = {}
>>> dict['a'] = 'alpha'  # assign key 'a' to have value 'alpha'
>>> dict['o'] = 'omega'
>>> dict['g'] = 'gamma'
## nice for debugging that you can dump out our data like this
>>> print dict
{'a': 'alpha', 'g': 'gamma', 'o': 'omega'}
>>> dict['a']  ## lookup key 'a'
'alpha'
>>> dict['x']  ## lookup key 'x' - error
Traceback (most recent call last):
  File "", line 1, in ?
KeyError: 'x'
>>> 'a' in dict  ## use in to test if key is present safely
True
>>> 'x' in dict
False
>>> dict.keys()  ## .keys() returns collection of keys in random order
['a', 'g', 'o']
>>> dict.values()
['alpha', 'gamma', 'omega']
>>> for k in dict.keys():  ## standard for loop over dict contents
  print k, '->', dict[k]
...
a -> alpha
g -> gamma
o -> omega
>>> dict.items()  ## dict contents as list of tuples (tuple is like a little list)
```

```
[('a', 'alpha'), ('g', 'gamma'), ('o', 'omega')]
```

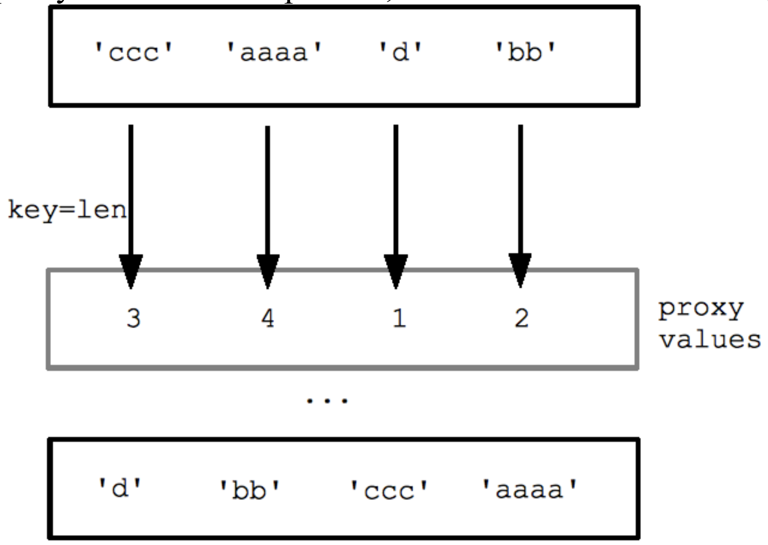
Sorting

The sorted(list) function takes in a list (or any linear sequence) and returns a new, sorted list of those elements. The optional reverse=True argument takes care of the common case where you want the sort reversed.

```
>>> a = ['baker', 'charlie', 'alpha']
>>> sorted(a)
['alpha', 'baker', 'charlie']
>>> a = [42, 6, 13]
>>> sorted(a)
[6, 13, 42]
>>> sorted(a, reverse=True)
[42, 13, 6]
```

Custom sorting

Python includes a new way to do custom sorting, and it's better than the old 2-argument cmp function way. Supply a function of *one argument*, a 'key' function, which computes a "proxy" value for each element in the original list, making a list of proxy values. The sorting algorithm then uses the proxy values for comparison, but does the sort on the original list.



This code uses the built-in len function as the key function, and so sorts the strings by length:

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

The key function can be defined using an ordinary def: so you can define a function that extracts the data you want to be used to guide the sort of the original list. Suppose you want to sort a bunch of strings by their 2nd letter.

```
  def second(word):
    return word[1]

  words = ['bc', 'zb', 'az']
  print sorted(words)  ## ['az', 'bc', 'zb']
  print sorted(words, key=second)  ## ['zb', 'bc', 'az']
```

Lambda (optional)

A lambda, in many languages, is a syntax for defining a function inline instead of in a separate def. The Python syntax is: lambda **var**: **expression**
Here's the above example written using lambda to define the key function inline:

```
  print sorted(words, key=lambda word: word[1])
```

Custom Sort + Dict Example

For the following example, a "tuple" works like a little list, but it is written within ( ) instead of [ ]. Suppose we have a dict of city->zip data and we want to print it out in order by zip. This can be done with custom sorting. The .items() on the dict returns the whole dict contents as a list of tuples. Each tuple is length 2, containing one key/value (city/zip) pair, like this:

```
  dict = ...
  items = dict.items()
  ## items is a list of length-2-tuples:
  ## [('palo alto', 94301), ('stanford', 94305), ('menlo park', 94025), ...]
```

We want to sort the items list into increasing order by zip code. Write a little function to use as the key function, taking in one element in the list (a tuple length 2), and returning the part we want to use in the sort (just the zip code):

```
def tuple_zip(tuple):
  """Returns zip from (city, zip) tuple."""
  return tuple[1]
```

Just use that as the key= function to sort the items list by zip:

```
  # change items to be sorted by zip
  items = sorted(items, key=tuple_zip)
  # Now can print them out
  for tuple in items:
    print tuple[0], tuple[1]
  # prints
  #   menlo park 94025
  #   palo alto 94301
  #   stanford 94305
```

Python Development Advice

The two most common newbie bugs are forgetting the ":" for an if or loop and having uneven indentation within an if or loop (this gets a compile-time error).

Since there's no separate compile step, it's very easy to make a little edit and then run to see how it works. Take advantage of that quickness in your development style. In particular, you can print out whatever your data structure is at some point and then sys.exit(0) to end the program. Keep doing that until it looks right, then work on getting the next data structure 8 lines lower to look right. Printing the concrete state of your data structure makes it easier to see the next step.

Good variable names are especially helpful in Python code, since they are the only thing there to remind you of what you have -- a string, a list, a dict. It's easy to lose track, so use variable names to keep things straight.