Python Cheat Sheet

When learning about a new technical area there's often a bewildering amount of detail to keep track of. Generations of techies have made use of cheat sheets to have an easy reference to these details.
A more detailed alternative is provided by the Python Quick Reference. If you want to print it out or save it locally several forms are available

Contents

Arithmetic

Constants (Number types):

- integers, e.g. 18, -341. A suffix L indicates a long integer, e.g. 34251673L.
- floating point values, e.g. 3.001

Operations:

- `+` for addition
- `-` for subtraction
- `*` for multiplication
- `/` for division (watch for integer division, e.g. 19 / 4 = 4, not 4.75)
- `%` for remainder or modulo, e.g. 19 % 4 = 3
- `**` for exponentiation, e.g. 2 ** 4 = 16

Strings

Constants: **`"This is a string"`**

Operations:

- `=` for assignment, e.g. **`name = "Tim Topper"`**
- `+` for concatenation, e.g. **`"Hi" + "Ho"`** --> **`"HiHo"`**
- `*` for repetition, e.g. **`3 * "Ho"`** --> **`"HoHoHo"`**

Notes:

- Strings can be delimited by either double quotes **`"Tim"`** or single quotes **`'Tim'`**.
- Multiline values can be assigned to a string by triple quoting the contents, e.g.

  ```
  silly = """Two
  Lines"""
  ```

  Trying the same thing without triple quoting results in an error, e.g.

  ```
  silly = "Two
  Lines"
  ```

- Another method is to embed control codes for newline characters into the string, e.g.

  ```
  silly = "Two\nLines"
  ```

  The most common control codes are: **`\n`** for newline and **`\t`** for tab. The Python Quick Reference provides a complete list.

Like lists Python strings are a sequence type so many list commands also work with strings.

- e.g. `s[ i ]` accesses element number `i` of the string `s`.

Testing string contents:

- `s.isalnum()`
- `s.isalpha()`
- `s.isdigit()`
- `s.isspace()`
- `s.islower()`
- `s.isupper()`
- `s.istitle()`
- `s.endswith( suffix )`
- `s.startswith( prefix )`

Finding things in strings:

- `s.count( substring )`
- `s.find( substring )`
- `s.index( substring )`
- `s.rfind( substring )`
- `s.rindex( substring )`

Changing strings. Remember that strings are immutable so to make a change "stick" you have to do, e.g. `s = s.title()`.

- `s.swapcase()`
- `s.upper()`
- `s.lower()`
- `s.title()`
- `s.capitalize()`
- `s.center()`
- `s.ljust()`
- `s.rjust()`
- `s.strip()`
- `s.lstrip( chars )`
- `s.rstrip( chars )`

String to list:

- `list = s.split( list )`

List to string:

- `s.join( list )` where s is the string to join the elements of list with.

Assignment

Use = to assign a name to a value, e.g. `distance = 48.1`, `name = "Tim Topper"`. Remember that the name has to be on the left hand side of the =, i.e. `48.1 = distance` is an error.

`print`

Use `print` to display stored values.

```
print list
```

Displays the value of each item in the list. Puts a space between each pair of values. Example: `print "The answer is", 5 + 2, "."` displays:

```
The answer is 7 .
```

Appending a comma to the end of a print statement holds the current output line open, e.g. the code

```
print "The answer is",
print 5 + 2, "."
```

displays

```
The answer is 7 .
```

on a single line.

For more control over output appearance embed formatting codes into output strings. See section 2.2.6.2 String Formatting Operations of the Python Library Reference for the gory details.

Getting Input

Use **`input`** to get numerical data from the user,

```
input( string )
```

and **`raw_input`** to get string data.

Both display *string* (if given) and then read a line of input, by default from the keyboard. The difference is that **`raw_input`** just returns the string, while **`input`** evaluates it as a Python exprssion and returns the result.

Example:

```
distance = input( "Enter the distance in miles: "
)
name = raw_input( "What is your name? " )
```

N.B. the spaces before the second `"` in each case.

`if`

Use if to execute one block of code or another, but not both.

```
if test:
    statements
elif test:
    statements
else:
    statements
```

N.B. the **`elif`** and **`else`** statements are optional as shown in the first two examples below.

Examples:

```
if x < 0: print x, "is negative"

if flip == 1:
    print "You got heads"
else:
    print "You got tails"

if num < 0:
    print "The number", num, "is negative."
elif num == 0:
    print "The number", num, "is neither positive nor negative."
else:
    print "The number", num, "is positive."
```

`while`

Use **`while`** to execute a block of code multiple times.

```
while test:
    statements
```

Examples:

```
x = 1
while x < 10
    print x
    x = x + 1
```

```
num = input( "Enter a number between 1 and 100: " )
while num < 1 or num > 100:
    print "Oops, your input value (", num, ") is out of range."
    num = input( "Be sure to enter a value between 1 and 100: " )
```

Play again? Repeating a program

```
again = "y"
while again == "y" or again == "Y" or again == "yes" or again == "Yes":

    #
    # Put the body of your program here
    #

    again = raw_input( "Play again (y/n)? " )

print "Thanks for playing"
```

Lists

Unlike many languages Python provides a built-in list type. A list constant is just a list of items separated by commas and placed inside square brackets, e.g. `[ "Tim", 42, "Molly" ]`.

Python provides for a wealth of list operations (complete list in reference manual):

- `list1 + list2` : concatenates `list1` and `list2`
- `list[ i ]` : access element number i in `list`.
- `len( list )` : returns the number of elements in `list`
- `del list[ i ]` : deletes element number i from `list`
- `list.append( value )` : appends `value` to `list`
- `list.sort()` : sorts the elements in `list`
- `list.reverse()` : reverses the order of the elements in `list`
- `list.index( value )` : returns the position of the first occurrence of `value` in `list`
- `list.insert( i, value )` : inserts `value` into `list` at position i
- `list.count( value )` : returns a count of the number of times `value` occurs in `list`
- `list.remove( value )` : deletes the first occurrence of `value` in `list`
- `list.pop()` : deletes and returns the last value in `list`
- `value in list` : is `True` if `value` occurs in list and `False` otherwise

N.B. the elements in lists are numbered from 0, **not** 1.

Text file processing

To read from a file a line at a time:

```
infilename = raw_input( "Name of file to read from: " )
infile = open( infilename, "r" )
for line in infile:
  # Do stuff with line.
  # Remember that line is a string even if it looks like a number,
  # e.g. num = int( line )
infile.close()
```

There's more than this of course. You can also read the entire file into a string in one fell swoop using `infile.read()`, read the entire file into a list of strings (one per line in the file) using `infile.readlines()`, or read a certain number of bytes using `infile.read( N )` where `N` gives the number of bytes to read.

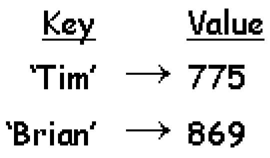
To write to a file:

```
outfilename = raw_input( "Name of file to write to: " )
outfile = open( outfilename, "w" )
print >> outfile, ...
outfile.close()
```

For more see the Python library reference on File Objects.

Dictionaries

Python provides a built-in lookup table type it calls a dictionary (often called hash tables in other languages).
A dictionary constant consists of a series of key-value pairs enclosed by curly braces, e.g. `d = { 'Tim' : 775, 'Brian' : 869 }`. This creates a dictionary we can visualize as:



Common dictionary operations include:

- `d[ 'Tim' ]` Accessing an element.
- `d[ 'A-S' ] = 770` Modifying or inserting a value.
- `d.has_key( 'Brian' )` Checking to see if there is a value for a particular key.
- `d.keys()` Get a list of all the keys in the dictionary. Often used for iterating through the entries in the dictionary.
- `d.values()` Get a list of all the values that occur in the dictionary.
- `del( d[ 'Tim' ] )` Delete an entry in the dictionary.
- `d.clear()` Delete all the entries in a dictionary.