Python basics Cheat Sheet

by stonekirby via cheatography.com/25026/cs/6414/

Primitive Datatypes and Operators

```
# You have numbers
3 # => 3
# Math is what you would expect
1 + 1 # => 2
8 - 1 # => 7
10 * 2 # => 20
# Except division which returns floats, real numbers, by default
35 / 5 # => 7.0
# Result of integer division truncated down both for positive and negative.
5 // 3 # => 1
5.0 // 3.0 # => 1.0 # works on floats too
-5 // 3 # => -2
-5.0 // 3.0 # => -2.0
# When you use a float, results are floats
3 * 2.0 # => 6.0
# Modulo operation
7 % 3 # => 1
# Exponentiation (x**y, x to the yth power)
2**4 # => 16
# Enforce precedence with parentheses
(1 + 3) * 2 # => 8
# Boolean values are primitives (Note: the capitalization)
True
False
# negate with not
not True # => False
not False # => True
# Boolean Operators
# Note "and" and "or" are case-sensitive
True and False # => False
False or True # => True
# Note using Bool operators with ints
0 and 2 # => 0
-5 or 0 # => -5
0 == False # => True
2 == True # => False
1 == True # => True
# Equality is ==
1 == 1 # => True
2 == 1 # => False
# Inequality is !=
1 != 1 # => False
2 != 1 # => True
```

Primitive Datatypes and Operators (cont)

```
# More comparisons
1 < 10 # => True
1 > 10 # => False
2 <= 2 # => True
2 >= 2 # => True
# Comparisons can be chained!
1 < 2 < 3 # => True
2 < 3 < 2 # => False
# (is vs. ==) is checks if two variable refer to the same object, but == checks
# if the objects pointed to have the same values.
a = [1, 2, 3, 4] # Point a at a new list, [1, 2, 3, 4]
b = a # Point b at what a is pointing to
b is a # => True, a and b refer to the same object
b == a # => True, a's and b's objects are equal
b = [1, 2, 3, 4] # Point a at a new list, [1, 2, 3, 4]
b is a # => False, a and b do not refer to the same object
b == a # => True, a's and b's objects are equal
```

Using [None]

```
# None is an object
None # => None
# Don't use the equality "==" symbol to compare objects to None
# Use "is" instead. This checks for equality of object identity.
"etc" is None # => False
None is None # => True
# None, 0, and empty strings/lists/dicts all evaluate to False.
# All other values are True
bool(0) # => False
bool("") # => False
bool([]) # => False
bool({}) # => False
```

Strings

```
# Strings are created with " or '
"This is a string."
'This is also a string.'
# Strings can be added too! But try not to do this.
"Hello " + "world!" # => "Hello world!"
# Strings can be added without using '+'
"Hello " "world!" # => "Hello world!"
# A string can be treated like a list of characters
"This is a string"[0] # => 'T'
# .format can be used to format strings, like this:
"{} can be {}".format("Strings", "interpolated") # => "Strings can be interpolated"
```

Strings (cont)

```
# You can repeat the formatting arguments to save some typing.
"{0} be nimble, {0} be quick, {0} jump over the {1}".format("Jack", "candle stick")
# => "Jack be nimble, Jack be quick, Jack jump over the candle stick"
# You can use keywords if you don't want to count.
"{name} wants to eat {food}".format(name="Bob", food="lasagna") # => "Bob wants to eat lasagna"
# If your Python 3 code also needs to run on Python 2.5 and below, you can also
# still use the old style of formatting:
"%s can be %s the %s way" % ("Strings", "interpolated", "old") # => "Strings can be interpolated the old way"
```

Lists

```
# Lists store sequences
li = []
# You can start with a prefilled list
other_li = [4, 5, 6]
# Add stuff to the end of a list with append
li.append(1) # li is now [1]
li.append(2) # li is now [1, 2]
li.append(4) # li is now [1, 2, 4]
li.append(3) # li is now [1, 2, 4, 3]
# Remove from the end with pop
li.pop() # => 3 and li is now [1, 2, 4]
# Let's put it back
li.append(3) # li is now [1, 2, 4, 3] again.
# Access a list like you would any array
li[0] # => 1
# Look at the last element
li[-1] # => 3
# Looking out of bounds is an IndexError
li[4] # Raises an IndexError
# You can look at ranges with slice syntax.
# (It's a closed/open range for you mathy types.)
li[1:3] # => [2, 4]
# Omit the beginning
li[2:] # => [4, 3]
# Omit the end
li[:3] # => [1, 2, 4]
# Select every second entry
li[::2] # =>[1, 4]
# Return a reversed copy of the list
li[::-1] # => [3, 4, 2, 1]
# Use any combination of these to make advanced slices
# li[start:end:step]
# Make a one layer deep copy using slices
li2 = li[:] # => li2 = [1, 2, 4, 3] but (li2 is li) will result in false.
```

Lists (cont)

```
# Remove arbitrary elements from a list with "del"
del li[2] # li is now [1, 2, 3]
# Remove first occurrence of a value
li.remove(2) # li is now [1, 3]
li.remove(2) # Raises a ValueError as 2 is not in the list
# Insert an element at a specific index
li.insert(1, 2) # li is now [1, 2, 3] again
# Get the index of the first item found matching the argument
li.index(2) # => 1
li.index(4) # Raises a ValueError as 4 is not in the list
# You can add lists
# Note: values for li and for other_li are not modified.
li + other_li # => [1, 2, 3, 4, 5, 6]
# Concatenate lists with "extend()"
li.extend(other_li) # Now li is [1, 2, 3, 4, 5, 6]
# Check for existence in a list with "in"
1 in li # => True
# Examine the length with "len()"
len(li) # => 6
```

Variables

```
# Python has a print function
print("I'm Python. Nice to meet you!") # => I'm Python. Nice to meet you!
# By default the print function also prints out a newline at the end.
# Use the optional argument end to change the end character.
print("Hello, World", end="!") # => Hello, World!
# Simple way to get input data from console
input_string_var = input("Enter some data: ") # Returns the data as a string
# Note: In earlier versions of Python, input() method was named as raw_input()
# No need to declare variables before assigning to them.
# Convention is to use lower_case_with_underscores
some_var = 5
some_var # => 5
# Accessing a previously unassigned variable is an exception.
# See Control Flow to learn more about exception handling.
some_unknown_var # Raises a NameError
```