

Python Cheat Sheet

Python is a beautiful language. It's easy to learn and fun, and its syntax is simple yet elegant. Python is a popular choice for beginners, yet still powerful enough to back some of the world's most popular products and applications from companies like NASA, Google, Mozilla, Cisco, Microsoft, and Instagram, among others. Whatever the goal, Python's design makes the programming experience feel almost as natural as writing in English.

Check out [Real Python](#) to learn more about Python and web development.

Note: This article applies to Python 2.7x specifically.

Email your questions or feedback to info@realpython.com.

1. Primitives

Numbers

Python has integers and floats. Integers are simply whole numbers, like 314, 500, and 716. Floats, meanwhile, are fractional numbers like 3.14, 2.867, 76.88887.

You can use the type method to check the value of an object.

```
>>> type(3)
<type 'int'>

>>> type(3.14)
<type 'float'>

>>> pi = 3.14
>>> type(pi)
<type 'float'>
```

In the above example, `pi` is the variable name, while `3.14` is the value.

You can use the basic mathematical operators:

```
>>> 3+3
6
>>> 3-3
0
>>> 3/3
1
>>> 3*3
9
>>> 3**3
27
>>> num = 3
>>> num = num - 1
>>> print num
2
>>> num = num + 10
>>> print num
12
>>> num += 10
>>> print num
22
>>> num -= 12
>>> print num
10
>>> num *= 10
>>> num
100
```

What happens when you divide 9 by 4?

```
>>> 9 / 4
2
```

What is the actual answer? 2 remainder 1 right? Or: 2.25

In Python 2.7x, when you divide a whole number by a whole number and the

answer is a fractional number, Python returns a whole number without the remainder. In other words, this type of division rounds the fraction down to the nearest whole number (commonly known as flooring the results).

Let's look at 12 divided by 5. If we want to get a fraction, we can just turn one of the numbers into a float.

```
>>> 12 / 5.0
2.4
>>> 12 / float(5)
2.4
```

There's also a special operator called a modulus, `%`, that returns the remainder after integer division.

```
>>> 10 % 3
1
>>> 10 / 3
3
```

One common use of modulus is determining if a number is divisible by another number. For example, we know that a number is even if it's divided by 2 and the remainder is 0.

```
>>> 10 % 2
0
>>> 12 % 2
0
```

Finally, make sure to use parentheses to enforce precedence.

```
>>> (2 + 3) * 5
25
```

Strings

Strings are used quite often in Python. Strings, are just that, a string of characters.

A character is anything you can type on the keyboard in one keystroke, like a letter, a number, or a backslash.

Python recognizes single and double quotes as the same thing, the beginning and ends of the strings.

```
>>> "string list"
'string list'
>>> 'string list'
'string list'
```

Now what if you have a quote in the middle of the string? Python needs help to recognize quotes as part of the English language and not as part of the Python language.

```
>>> "I can't do that"
"I can't do that"
>>> "He said \"no\" to me"
"He said \"no\" to me"
```

Now you can also join strings with use of variables as well.

```
>>> a = "first"
>>> b = "last"
>>> a + b
'firstlast'
```

If you want a space in between, you can change `a` to the word with a space after.

```
>>> a = "first "
>>> a + b
"first last"
```

There are also different string methods for you to choose from as well - like `upper`, `lower`, `replace`, and `count`.

Upper does just what it sounds like, changes your string to have uppercase

letters.

```
>>> w='woah! '  
>>> w.upper()  
'WOAH! '
```

Lower is just that as well, keeping your string in lower case letters.

```
>>> w='WOAH! '  
>>> w.lower()  
'woah! '
```

Replace allows you to replace any character with another character.

```
>>>r='rule'  
>>>r.replace('r','m')  
'mule'
```

Count lets you know how many times x appears in the string (can be numbers or a string of words as well).

```
>>>numbers=['1','2','1','2','2']  
>>>numbers.count('2')  
3
```

You can also format strings with the format method.

```
>>> "{0} is a lot of {1}".format("Python", "fun!")  
'Python is a lot of fun!'
```

Booleans

Boolean values are simply **True** or **False**.

Check to see if a value is equal to another value with two equal signs.

```
>>> 10 == 10
True
>>> 10 == 11
False
>>> "jack" == "jack"
True
>>> "jack" == "jake"
False
```

To check for inequality use `!=`.

```
>>> 10 != 10
False
>>> 10 != 11
True
>>> "jack" != "jack"
False
>>> "jack" != "jake"
True
```

You can also test for `>`, `<`, `>=`, and `<=`.

```
>>> 10 > 10
False
>>> 10 < 11
True
>>> 10 >= 10
True
>>> 10 <= 11
True
>>> 10 <= 10 < 0
False
>>> 10 <= 10 < 11
True
>>> "jack" > "jack"
False
>>> "jack" >= "jack"
True
```

2. Collections

Lists

Lists are containers for holding values.

```
>>> fruits = ['apple','lemon','orange','grape']
>>> fruits
['apple', 'lemon', 'orange', 'grape']
```

To access the elements in the list you can use the placement in the list as an indicator. This means numbering the items aligned with their placement in the list. But, you must remember that the list starts with 0.

```
>>> fruits[2]
'orange'
```

If the list is longer and you need to count from the end you can also do that.

```
>>> fruits[-2]
'orange'
```

Now, sometimes lists can get long and you want to keep track of how many elements you have in your list. To find this, use `len` which is the length function.

```
>>> len(fruits)
4
```

Use `append` to add a new object to the end of the list and `pop` to remove objects from the end.

```
>>> fruits.append('blueberry')
>>> fruits
['apple', 'lemon', 'orange', 'grape', 'blueberry']
>>> fruits.append('tomato')
>>> fruits
['apple', 'lemon', 'orange', 'grape', 'blueberry', 'tomato']
>>> fruits.pop()
'tomato'
>>> fruits
['apple', 'lemon', 'orange', 'grape', 'blueberry']
```

Check to see if a value exists using `in`.

```
>>> 'apple' in fruits
True
>>> 'tomato' in fruits
False
```

Dictionaries

A dictionary optimizes element lookups. It uses keys and values, instead of numbers as placeholders. Each key must have a value. You can use a word to look up a value.

```
>>> words={'apple':'red','lemon':'yellow'}
>>> words
{'lemon': 'yellow', 'apple': 'red'}
>>> words['apple']
'red'
>>> words['lemon']
'yellow'
```

This will also work with numbers.


```
>>> dict={'one':1,'two':2}
>>> dict
{'two': 2, 'one': 1}
>>>
```

Output all the keys as a list with `keys()` and all the values with `values()`.

```
>>> words.keys()
['lemon', 'apple']
>>> words.values()
['yellow', 'red']
```

3. Control Statements

IF Statements

The IF statement is used to check if a condition is true.

Essentially, if the condition is true, the Python interpreter runs a block of statements called the if-block. If the statement is false, the interpreter skips the if-block and processes another block of statements called the else-block. The else clause is optional.

Let's look at two quick examples.

```
>>> num = 20
>>> if num == 20:
...     print 'the number is 20'
... else:
...     print 'the number is not 20'
...
the number is 20

>>> num = 21
>>> if num == 20:
...     print 'the number is 20'
... else:
...     print 'the number is not 20'
...
the number is not 20
```

You can also add an `elif` clause to add another condition.

```
>>> num = 21
>>> if num == 20:
...     print 'the number is 20'
... elif num > 20:
...     print 'the number is greater than 20'
... else:
...     print 'the number is less than 20' ...
the number is greater than 20
```

Loops

There are 2 kinds of loops used in Python. The For loop and the While loop. For loops are traditionally used when you have a piece of code which you want to repeat n number of times. They are also commonly used to loop or iterate over lists.

```
>>> colors = ('red', 'blue', 'green')
>>> colors
('red', 'blue', 'green')
>>> for favorite in colors:
...     print "I love " + favorite ...
I love red
I love blue
I love green
```

While loops, like the For Loop, are used for repeating sections of code - but unlike a for loop, the while loop will not run n times, but until a defined condition is met.

```
>>> num = 1
>>> num
1
>>> while num <=5:
...     print num
...     num += 1
...
1
2
3
4
5
```

4. Functions

Functions are blocks of reusable code that perform a single task.

You use `def` to define (or create) a new function then you call a function by adding parameters to the function name.

```
>>> def multiply(num1, num2):  
...     return num1 * num2  
...  
>>> multiply(2,2)  
4
```

You can also set default values for parameters.

```
>>> def multiply(num1, num2=10):  
...     return num1 * num2  
...  
>>> multiply(2)  
20
```

Ready to learn more?

[—————](#)