Real Python: Python 3 Cheat Sheet

Contents

1 Introduction **2**

2 Primitives **3**

Numbers 3

Strings 5

Booleans 7

3 Collections **9**

Lists 9

Dictionaries 11

4 Control Statements **12**

IF Statements 12

Loops 14

5 Functions **15**

Chapter 1

Introduction

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

Chapter 2

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

In the last 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
1.0
>>> 3 / 2
```

```
1.5
>>> 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
```

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

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

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
>>> 2 + 3 * 5
17
```

Strings

Strings are used quite often in Python. Strings, are just that, a string of characters - which s 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 end of the strings.

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

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 'cant do that"
'I 'cant do that'
>>> "He said \"no\" to me"
'He said "no" to me'
```

Now you can also join (concatenate) 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 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 all uppercase letters.

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

Can you guess what `lower()` does?

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

`replace()` allows you to replace any character with another character.

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

Finally, `count()` lets you know how many times a certain character appears in the string.

```
>>> number_list =['one', 'two', 'one', 'two', 'two']
>>> number_list.count('two')
3
```

You can also format/create 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
```

Chapter 3

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 their associated index value. Just remember that the list starts with `0`, not `1`.

```
>>> fruits[2]‘’
orange
```

If the list is long and you need to count from the end you can do that as well.

```
>>> 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 the `len()` function.

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

Use `append()` to add a new element to the end of the list and `pop()` to remove an element 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 the list.

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

Dictionaries

A dictionary optimizes element lookups. It uses key/value pairs, instead of numbers as placeholders. Each key must have a value, and you can use a key to look up a value.

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

This will also work with numbers.

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

Output all the keys with `keys()` and all the values with `values()`.

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

Chapter 4

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 to check for.

```
>>> 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', 'green', 'blue']
>>> colors
['red', 'green', 'blue']
>>> for color in colors:
...     print('I love ' + color)
...
I love red
I love green
I love blue
```

`while` loops, like the `for` Loop, are used for repeating sections of code - but unlike a `for` loop, the `while` loop continues until a defined condition is met.

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

Chapter 5

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? Visit **Real Python** to learn Python and web development. Cheers!

***args** haben 2 Anwendungsfälle:

1. Wenn die Parameterwerte für eine Funktion in Objektform vorliegen (einer Liste oder einem Tupel) können sie eben in dieser Form an die Funktion übergeben werden.
2. Die Anzahl der Übergabeparameter an die Funktion soll variabel bleiben.

1. Übergabe von Parametern in Objektform

- Übergabe der Parameter als Tupel oder Liste. Hier werden die Elemente *In-Order-Of-Appearance* den Parametern der Funktion zugeordnet.

```
def myFunction(a,b,c,*args):
    return print(a,b,c, args)

myFunction(*(1,2,3,4)) # Übergabe als Tupel
myFunction(*[1,2,3,4]) # Übergabe als Liste
>>1, 2, 3 (4,)
```

In-Order-Of-Appearence wurden die Werte 1,2,3 den Parametern a,b,c zugeordnet. Der verbleibende Wert 4, verbleibt ungenutzt als Tupel innerhalb der Funktion und wird über den args-Parameternamen ansprechbar.

Durch den Funktionsaufruf mit *args verkürzt sich gegenüber einem Aufruf mit Parameternamen auch der Quellcode:

```
param = [1,2,3]
myFunction(*param) # Aufruf mit args
myFunction(a=1,b=2,c=3) # Aufruf mit Parameternamen
```

2. Variable Länge der Übergabeparameter

Eine weitere Option die durch den *args-Parameter entsteht, ist die beliebige Fortschreibung von Übergaben an die Funktion. Hier werden im ersten Beispiel der Funktion myFunction 7 Werte übergeben. Die Werte 1,2,3 werden den Parametern a,b,c zugeordnet. Die weiteren 4 Werte werden (wie bereits oben gesehen) als Tupel in der Funktion ansprechbar. Im zweiten Beispiel sind die Keyword-Argumente mit Werten besetzt und das *args-Tupel steht so wie es ist, also mit den Werten 4,5,6,7 , in der Funktion zur Verfügung.

```
myFunction(1,2,3,4,5,6,7)
>>1, 2, 3 (4, 5, 6, 7)

myFunction(1,2,3,(4,5,6,7))
>>1 2 3 ((4, 5, 6, 7),)
```

****kwargs**

- Übergabe der Parameter als Dictionary. Über die Keys werden die Werte den Parametern der Funktion zugeordnet.

**kwargs sind verglichen mit *args eine konservative Variante an die Funktion ein Objekt zu übergeben. Da das Objekt ein Dictionary ist, verhindert man eine Falschzuordnung von

Parameter zu Wert wie es durch die *Order-Of-Appearence*-Regel bei der Verwendung *args passieren kann.

```
def myFunction(a,b,c,**kwargs):
    return print(a,b,c, **kwargs)

myDict = {'b':2,'a':1,'c':3}
myFunction(**myDict)
```

Bei der Nutzung von **kwargs werden alle benötigten Parameter mit ihren Werten als Key-Value Paare in das Dictionary eingetragen und der Funktion übergeben. Enthält das Dictionary Keys, die nicht den Parametern der Funktion zuzuordnen sind, führt dies zu einem Fehler:

```
def myFunction(a,b,c,**kwargs):
    return print(a,b,c, **kwargs)

myDict = {'b':2,'a':1,'c':3,'d':4}
myFunction(**myDict)
```

TypeError: ‚d‘ is an invalid keyword argument for this function

*args und **kwargs gemeinsam in einer Funktion

Natürlich lassen sich *args und **kwargs auch gemeinsam in einer Funktion verwenden. Wir sehen an dem Beispielcode unten 5 Funktionsaufrufe von myFunction, die in ihrer Definition sowohl *args als auch **kwargs enthält.

```
def myFunction(parameter_1, parameter_2, *args ,**kwargs):
    print(parameter_1)
    print(parameter_2)
    print(*args)


myFunction('A',*(100,200,300))
>>A
>>100
>>(200, 300)

myFunction('A',**{'parameter_2':1000})
>>A
>>1000
>>()

myFunction(parameter_2 = 1000, *('A',))
>>A
>>1000
>>()

myFunction('A',**{'parameter_1':1000,'parameter_2':1000})
```

TypeError: myFunction() got multiple values for argument ‚parameter_1‘

```
myFunction('A',*(1,2,3,),**{'parameter_2':1000})
```

TypeError: myFunction() got multiple values for argument ‚parameter_2‘

Real Python Cheat Sheet

realpython.com

Python Decorators Examples

Learn more about decorators in Python in our in-depth tutorial at realpython.com/primer-on-python-decorators/

Using decorators

The normal way of using a decorator is by specifying it just before the definition of the function you want to decorate:

```
@decorator
def f(arg_1, arg_2):
    ...
```

If you want to decorate an already existing function you can use the following syntax:

```
f = decorator(f)
```

Decorator not changing the decorated function

If you don't want to change the decorated function, a decorator is simply a function taking in and returning a function:

```
def name(func):
    # Do something with func
    return func
```

Example: Register a list of decorated functions.

```
def register(func):
    """Register a function as a plug-in"""
    PLUGINS[func.__name__] = func
    return func
```

Basic decorator

Template for basic decorator that can modify the decorated function:

```
import functools

def name(func):
    @functools.wraps(func)
    def wrapper_name(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapper_name
```

Example: A `timer` decorator that prints the runtime of a function.

```
import functools
import time

def timer(func):
    """Print the runtime of the function"""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()
        value = func(*args, **kwargs)
        end_time = time.perf_counter()
        run_time = end_time - start_time
        print(f"Run time: {run_time:.4f} secs")
        return value
    return wrapper_timer
```

Decorator with arguments

If you want your decorator to take arguments, create a decorator factory that can create decorators:

```
import functools

def name(arg_1, ...):
    def decorator_name(func):
        @functools.wraps(func)
        def wrapper_name(*args, **kwargs):
            # Do something before using arg_1, ...
            value = func(*args, **kwargs)
            # Do something after using arg_1, ...
            return value
        return wrapper_name
    return decorator_name
```

Example: Rate limit your code by sleeping a given amount of seconds before calling the function.

```
import functools
import time

def slow_down(rate):
    """Sleep before calling the function"""
    def decorator_slow_down(func):
        @functools.wraps(func)
        def wrapper_slow_down(*args, **kwargs):
            time.sleep(rate)
            return func(*args, **kwargs)
        return wrapper_slow_down
    return decorator_slow_down
```

Decorators that can optionally take arguments

If you want your decorator to be able to be called with or without arguments, you need a dummy argument, _func, that is set automatically if the decorator is called without arguments:

```
import functools

def name(_func=None, *, arg_1=val_1, ...):
    def decorator_name(func):
        @functools.wraps(func)
        def wrapper_name(*args, **kwargs):
            # Do something before using arg_1, ...
            value = func(*args, **kwargs)
            # Do something after using arg_1,  ...
            return value
        return wrapper_name

    if _func is None:
        return decorator_name
    else:
        return decorator_name(_func)
```

Example: Rate limit your code by sleeping an optionally given amount of seconds before calling the function.

```
import functools
import time

def slow_down(_func=None, *, rate=1):
    """Sleep before calling the function"""
    def decorator_slow_down(func):
        @functools.wraps(func)
        def wrapper_slow_down(*args, **kwargs):
            time.sleep(rate)
            return func(*args, **kwargs)
        return wrapper_slow_down

    if _func is None:
        return decorator_slow_down
    else:
        return decorator_slow_down(_func)
```

Decorators that keep state

If you need your decorator to maintain state, use a class as a decorator:

```
import functools

class Name:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func
        # Initialize state attributes

    def __call__(self, *args, **kwargs):
        # Update state attributes
        return self.func(*args, **kwargs)
```

Example: Count the number of times the decorated function is called.

```
import functools

class CountCalls:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func
        self.num_calls = 0

    def __call__(self, *args, **kwargs):
        self.num_calls += 1
        print(f"Call {self.num_calls}")
        return self.func(*args, **kwargs)
```