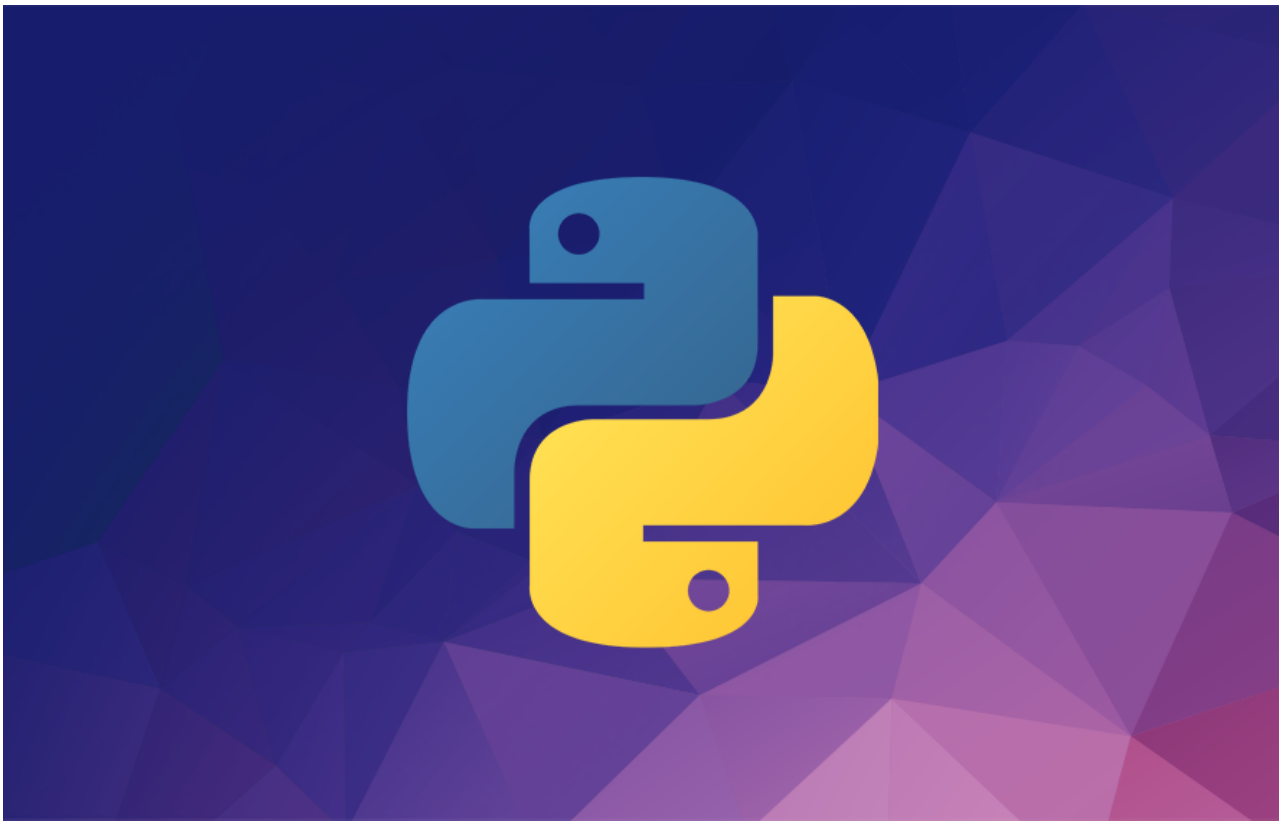
Python Cheat Sheet

Mosh Hamedani



Code with Mosh (codewithmosh.com)

1st Edition

About this Cheat Sheet

This cheat sheet includes the materials I've covered in my Python tutorial for Beginners on YouTube. Both the YouTube tutorial and this cheat cover the core language constructs but they are not complete by any means.

If you want to learn everything Python has to offer and become a Python expert, check out my Complete Python Programming Course:

http://bit.ly/complete-python-course

About the Author

Hi! My name is Mosh Hamedani. I'm a software engineer with two decades of experience and I've taught over three million how to code or how to become a professional software engineer. It's my mission to make software engineering simple and accessible to everyone.

https://codewithmosh.com

https://youtube.com/user/programmingwithmosh

https://twitter.com/moshhamedani

https://facebook.com/programmingwithmosh/

Variables

We use variables to temporarily store data in computer's memory.

```
price = 10

rating = 4.9

course_name = 'Python for Beginners'

is_published = True
```

In the above example,

- **price** is an *integer* (a whole number without a decimal point)
- **rating** is a *float* (a number with a decimal point)
- **course_name** is a *string* (a sequence of characters)
- **is_published** is a *boolean*. Boolean values can be True or False.

Comments

We use comments to add notes to our code. Good comments explain the hows and whys, not what the code does. That should be reflected in the code itself. Use comments to add reminders to yourself or other developers, or also explain your assumptions and the reasons you've written code in a certain way.

```
# This is a comment and it won't get executed.
# Our comments can be multiple lines.
```

Receiving Input

We can receive input from the user by calling the **input()** function.

```
birth_year = int(input('Birth year: '))
```

The **input()** function always returns data as a string. So, we're converting the result into an integer by calling the built-in **int()** function.

Strings

We can define strings using single (‘ ‘) or double (“ “) quotes.

To define a multi-line string, we surround our string with tripe quotes (“””).

We can get individual characters in a string using square brackets [].

```
course = ‘Python for Beginners’
course[0]   # returns the first character
course[1]   # returns the second character
course[-1]  # returns the first character from the end
course[-2]  # returns the second character from the end
```

We can slice a string using a similar notation:

```
course[1:5]
```

The above expression returns all the characters starting from the index position of 1 to 5 (but excluding 5). The result will be **ytho**

If we leave out the start index, 0 will be assumed.

If we leave out the end index, the length of the string will be assumed.

We can use formatted strings to dynamically insert values into our strings:

```
name = ‘Mosh’

message = f’Hi, my name is {name}’


message.upper()   # to convert to uppercase

message.lower()   # to convert to lowercase

message.title()   # to capitalize the first letter of every word

message.find(‘p’) # returns the index of the first occurrence of p
                   (or -1 if not found)

message.replace(‘p’, ‘q’)
```

To check if a string contains a character (or a sequence of characters), we use the **in** operator:

```
contains = 'Python' in course
```

Arithmetic Operations

```
+

-

*

/     # returns a float

//    # returns an int

%     # returns the remainder of division

**    # exponentiation - x ** y = x to the power of y
```

Augmented assignment operator:

```
x = x + 10

x += 10
```

Operator precedence:

1. parenthesis
2. exponentiation
3. multiplication / division
4. addition / subtraction

If Statements

```
if is_hot:
    print("hot day")
elif is_cold:
    print("cold day")
else:
    print("beautiful day")
```

Logical operators:

```
if has_high_income and has_good_credit:
    ...
if has_high_income or has_good_credit:
    ...
is_day = True
is_night = not is_day
```

Comparison operators

```
a > b
a >= b (greater than or equal to)
a < b
a <= b
a == b (equals)
a != b (not equals)
```

While loops

```
i = 1
while i < 5:
    print(i)
    i += 1
```

For loops

```
for i in range(1, 5):
    print(i)
```

- **range(5):** generates 0, 1, 2, 3, 4
- **range(1, 5):** generates 1, 2, 3, 4
- **range(1, 5, 2):** generates 1, 3

Lists

```
numbers = [1, 2, 3, 4, 5]
numbers[0]            # returns the first item
numbers[1]            # returns the second item
numbers[-1]           # returns the first item from the end
numbers[-2]           # returns the second item from the end


numbers.append(6)     # adds 6 to the end
numbers.insert(0, 6) # adds 6 at index position of 0
numbers.remove(6)     # removes 6
numbers.pop()         # removes the last item
numbers.clear()       # removes all the items
numbers.index(8)      # returns the index of first occurrence of 8
numbers.sort()        # sorts the list
numbers.reverse()     # reverses the list
numbers.copy()        # returns a copy of the list
```

Tuples

They are like read-only lists. We use them to store a list of items. But once we define a tuple, we cannot add or remove items or change the existing items.

```
coordinates = (1, 2, 3)
```

We can unpack a list or a tuple into separate variables:

```
x, y, z = coordinates
```

Dictionaries

We use dictionaries to store key/value pairs.

```
customer = {
    "name": "John Smith",
    "age": 30,
    "is_verified": True
}
```

We can use strings or numbers to define keys. They should be unique. We can use any types for the values.

```
customer["name"]                # returns "John Smith"
customer["type"]                # throws an error
customer.get("type", "silver") # returns "silver"
customer["name"] = "new name"
```

Functions

We use functions to break up our code into small chunks. These chunks are easier to read, understand and maintain. If there are bugs, it's easier to find bugs in a small chunk than the entire program. We can also re-use these chunks.

```
def greet_user(name):
    print(f"Hi {name}")


greet_user("John")
```

Parameters are placeholders for the data we can pass to functions. **Arguments** are the actual values we pass.

We have two types of arguments:

- Positional arguments: their position (order) matters
- Keyword arguments: position doesn't matter - we prefix them with the parameter name.

```
# Two positional arguments
greet_user("John", "Smith")

# Keyword arguments
calculate_total(order=50, shipping=5, tax=0.1)
```

Our functions can return values. If we don't use the return statement, by default **None** is returned. None is an object that represents the absence of a value.

```
def square(number):
    return number * number

result = square(2)
print(result)  # prints 4
```

Exceptions

Exceptions are errors that crash our programs. They often happen because of bad input or programming errors. It's our job to anticipate and handle these exceptions to prevent our programs from cashing.

```
try:
    age = int(input('Age: '))
    income = 20000
    risk = income / age
    print(age)
except ValueError:
    print('Not a valid number')
except ZeroDivisionError:
    print('Age cannot be 0')
```

Classes

We use classes to define new types.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def move(self):
        print("move")
```

When a function is part of a class, we refer to it as a **method**.

Classes define templates or blueprints for creating objects. An object is an instance of a class. Every time we create a new instance, that instance follows the structure we define using the class.

```
point1 = Point(10, 5)
point2 = Point(2, 4)
```

__init__ is a special method called constructor. It gets called at the time of creating new objects. We use it to initialize our objects.

Inheritance

Inheritance is a technique to remove code duplication. We can create a *base class* to define the common methods and then have other classes inherit these methods.

```
class Mammal:
  def walk(self):
     print("walk")

class Dog(Mammal):
  def bark(self):
     print("bark")


dog = Dog()
dog.walk()   # inherited from Mammal
dog.bark()   # defined in Dog
```

Modules

A module is a file with some Python code. We use modules to break up our program into multiple files. This way, our code will be better organized. We won't have one gigantic file with a million lines of code in it!

There are 2 ways to import modules: we can import the entire module, or specific objects in a module.

```
# importing the entire converters module
import converters
converters.kg_to_lbs(5)

# importing one function in the converters module
from converters import kg_to_lbs
kg_to_lbs(5)
```

Packages

A package is a directory with **__init__.py** in it. It can contain one or more modules.

```
# importing the entire sales module
from ecommerce import sales
sales.calc_shipping()

# importing one function in the sales module
from ecommerce.sales import calc_shipping
calc_shipping()
```

Python Standard Library

Python comes with a huge library of modules for performing common tasks such as sending emails, working with date/time, generating random values, etc.

Random Module

```
import random

random.random()       # returns a float between 0 to 1
random.randint(1, 6)  # returns an int between 1 to 6

members = ['John', 'Bob', 'Mary']
leader = random.choice(members) # randomly picks an item
```

Pypi

Python Package Index (pypi.org) is a directory of Python packages published by Python developers around the world. We use **pip** to install or uninstall these packages.

```
pip install openpyxl
```

```
pip uninstall openpyxl
```

Want to Become a Python Expert?

If you're serious about learning Python and getting a job as a Python developer, I highly encourage you to enroll in my Complete Python Course. Don't waste your time following disconnected, outdated tutorials. My Complete Python Course has everything you need in one place:

- 12 hours of HD video
- Unlimited access - watch it as many times as you want
- Self-paced learning - take your time if you prefer
- Watch it online or download and watch offline
- Certificate of completion - add it to your resume to stand out
- 30-day money-back guarantee - no questions asked

The price for this course is $149 but the first 200 people who have downloaded this cheat sheet can get it for $14.99 using the coupon code **CHEATSHEET**:

http://bit.ly/complete-python-course

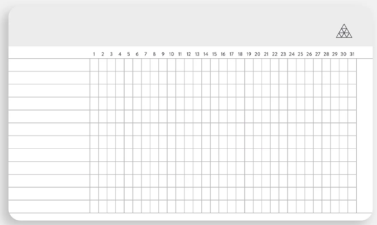
HOW TO CREATE A GOOD HABIT

The 1st Law	Make It Obvious
1.1	Fill out the Habits Scorecard. Write down your current habits to become aware of them.
1.2	Use implementation intentions: "I will [BEHAVIOR] at [TIME] in [LOCATION]."
1.3	Use habit stacking: "After [CURRENT HABIT], I will [NEW HABIT]."
1.4	Design your environment. Make the cues of good habits obvious and visible.
The 2nd Law	**Make It Attractive**
2.1	Use temptation bundling. Pair an action you want to do with an action you need to do.
2.2	Join a culture where your desired behavior is the normal behavior.
2.3	Create a motivation ritual. Do something you enjoy immediately before a difficult habit.
The 3rd Law	**Make It Easy**
3.1	Reduce friction. Decrease the number of steps between you and your good habits.
3.2	Prime the environment. Prepare your environment to make future actions easier.
3.3	Master the decisive moment. Optimize the small choices that deliver outsized impact.
3.4	Use the Two-Minute Rule. Downscale your habits until they can be done in two minutes or less.
3.5	Automate your habits. Invest in technology and onetime purchases that lock in future behavior.
The 4th Law	**Make It Satisfying**
4.1	Use reinforcement. Give yourself an immediate reward when you complete your habit.
4.2	Make "doing nothing" enjoyable. When avoiding a bad habit, design a way to see the benefits.
4.3	Use a habit tracker. Keep track of your habit streak and "don't break the chain."
4.4	Never miss twice. When you forget to do a habit, make sure you get back on track immediately.

HOW TO BREAK A BAD HABIT

Inversion of the 1st Law	Make It Invisible
1.5	Reduce exposure. Remove the cues of your bad habits from your environment.
Inversion of the 2nd Law	**Make It Unattractive**
2.4	Reframe your mindset. Highlight the benefits of avoiding your bad habits.
Inversion of the 3rd Law	**Make It Difficult**
3.6	Increase friction. Increase the number of steps between you and your bad habits.
3.7	Use a commitment device. Restrict your future choices to the ones that benefit you.
Inversion of the 4th Law	**Make It Unsatisfying**
4.5	Get an accountability partner. Ask someone to watch your behavior.
4.6	Create a habit contract. Make the costs of your bad habits public and painful.

Enjoyed Atomic Habits? Here are a few other habit-building products you might like.



HABIT TRACKER CARDS

Habit trackers are an easy and effective way to visualize your progress and motivate you to show up again tomorrow. I recommend these habit tracker index cards from Baronfig. They are easy to use and can be displayed on your desk, fridge, or anywhere you want to keep your habits top of mind.

jamesclear.com/cards

ATOMIC HABITS ENGRAVED PENS

These laser engraved pens feature popular quotes from Atomic Habits. These pens won't build better habits for you, but they will remind you of the core principles mentioned in Atomic Habits. Each pen is refillable and designed to last a lifetime. There are 3 different engraving options: grab your favorite (or get the bundle pack with all three).

jamesclear.com/pens

Use the code **HABITS20** to get 20% off these items at Baronfig.com