

Beginner's Python Cheat Sheet

Variables and Strings

Variables are used to assign labels to values. A string is a series of characters, surrounded by single or double quotes. Python's f-strings allow you to use variables inside strings to build dynamic messages.

Hello world

```
print("Hello world!")
```

Hello world with a variable

```
msg = "Hello world!"  
print(msg)
```

f-strings (using variables in strings)

```
first_name = 'albert'  
last_name = 'einstein'  
full_name = f"{first_name} {last_name}"  
print(full_name)
```

Lists

A list stores a series of items in a particular order. You access items using an index, or within a loop.

Make a list

```
bikes = ['trek', 'redline', 'giant']
```

Get the first item in a list

```
first_bike = bikes[0]
```

Get the last item in a list

```
last_bike = bikes[-1]
```

Looping through a list

```
for bike in bikes:  
    print(bike)
```

Adding items to a list

```
bikes = []  
bikes.append('trek')  
bikes.append('redline')  
bikes.append('giant')
```

Making numerical lists

```
squares = []  
for x in range(1, 11):  
    squares.append(x**2)
```

Lists (cont.)

List comprehensions

```
squares = [x**2 for x in range(1, 11)]
```

Slicing a list

```
finishers = ['sam', 'bob', 'ada', 'bea']  
first_two = finishers[:2]
```

Copying a list

```
copy_of_bikes = bikes[:]
```

Tuples

Tuples are similar to lists, but the items in a tuple can't be modified.

Making a tuple

```
dimensions = (1920, 1080)  
resolutions = ('720p', '1080p', '4K')
```

If statements

If statements are used to test for particular conditions and respond appropriately.

Conditional tests

equal	x == 42
not equal	x != 42
greater than	x > 42
or equal to	x >= 42
less than	x < 42
or equal to	x <= 42

Conditional tests with lists

```
'trek' in bikes  
'surly' not in bikes
```

Assigning boolean values

```
game_active = True  
can_edit = False
```

A simple if test

```
if age >= 18:  
    print("You can vote!")
```

If-elif-else statements

```
if age < 4:  
    ticket_price = 0  
elif age < 18:  
    ticket_price = 10  
elif age < 65:  
    ticket_price = 40  
else:  
    ticket_price = 15
```

Dictionaries

Dictionaries store connections between pieces of information. Each item in a dictionary is a key-value pair.

A simple dictionary

```
alien = {'color': 'green', 'points': 5}
```

Accessing a value

```
print(f"The alien's color is {alien['color']}.")
```

Adding a new key-value pair

```
alien['x_position'] = 0
```

Looping through all key-value pairs

```
fav_numbers = {'eric': 7, 'ever': 4, 'erin': 47}
```

```
for name, number in fav_numbers.items():  
    print(f"{name} loves {number}.")
```

Looping through all keys

```
fav_numbers = {'eric': 7, 'ever': 4, 'erin': 47}
```

```
for name in fav_numbers.keys():  
    print(f"{name} loves a number.")
```

Looping through all the values

```
fav_numbers = {'eric': 7, 'ever': 4, 'erin': 47}
```

```
for number in fav_numbers.values():  
    print(f"{number} is a favorite.")
```

User input

Your programs can prompt the user for input. All input is stored as a string.

Prompting for a value

```
name = input("What's your name? ")  
print(f"Hello, {name}!")
```

Prompting for numerical input

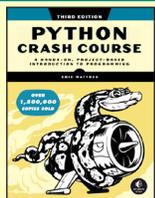
```
age = input("How old are you? ")  
age = int(age)
```

```
pi = input("What's the value of pi? ")  
pi = float(pi)
```

Python Crash Course

*A Hands-on, Project-Based
Introduction to Programming*

ehmatthes.github.io/pcc_3e



While loops

A while loop repeats a block of code as long as a certain condition is true. While loops are especially useful when you can't know ahead of time how many times a loop should run.

A simple while loop

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

Letting the user choose when to quit

```
msg = ''
while msg != 'quit':
    msg = input("What's your message? ")
    print(msg)
```

Functions

Functions are named blocks of code, designed to do one specific job. Information passed to a function is called an argument, and information received by a function is called a parameter.

A simple function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

Passing an argument

```
def greet_user(username):
    """Display a personalized greeting."""
    print(f"Hello, {username}!")

greet_user('jesse')
```

Default values for parameters

```
def make_pizza(topping='pineapple'):
    """Make a single-topping pizza."""
    print(f"Have a {topping} pizza!")

make_pizza()
make_pizza('mushroom')
```

Returning a value

```
def add_numbers(x, y):
    """Add two numbers and return the sum."""
    return x + y

sum = add_numbers(3, 5)
print(sum)
```

Classes

A class defines the behavior of an object and the kind of information an object can store. The information in a class is stored in attributes, and functions that belong to a class are called methods. A child class inherits the attributes and methods from its parent class.

Creating a dog class

```
class Dog:
    """Represent a dog."""

    def __init__(self, name):
        """Initialize dog object."""
        self.name = name

    def sit(self):
        """Simulate sitting."""
        print(f"{self.name} is sitting.")

my_dog = Dog('Peso')

print(f"{my_dog.name} is a great dog!")
my_dog.sit()
```

Inheritance

```
class SARDog(Dog):
    """Represent a search dog."""

    def __init__(self, name):
        """Initialize the sardog."""
        super().__init__(name)

    def search(self):
        """Simulate searching."""
        print(f"{self.name} is searching.")

my_dog = SARDog('Willie')

print(f"{my_dog.name} is a search dog.")
my_dog.sit()
my_dog.search()
```

Infinite Skills

If you had infinite programming skills, what would you build?

As you're learning to program, it's helpful to think about the real-world projects you'd like to create. It's a good habit to keep an "ideas" notebook that you can refer to whenever you want to start a new project.

If you haven't done so already, take a few minutes and describe three projects you'd like to create. As you're learning you can write small programs that relate to these ideas, so you can get practice writing code relevant to topics you're interested in.

Working with files

Your programs can read from files and write to files. The `pathlib` library makes it easier to work with files and directories. Once you have a path defined, you can work with the `read_text()` and `write_text()` methods.

Reading the contents of a file

The `read_text()` method reads in the entire contents of a file. You can then split the text into a list of individual lines, and then process each line as you need to.

```
from pathlib import Path

path = Path('siddhartha.txt')
contents = path.read_text()
lines = contents.splitlines()

for line in lines:
    print(line)
```

Writing to a file

```
path = Path('journal.txt')

msg = "I love programming."
path.write_text(msg)
```

Exceptions

Exceptions help you respond appropriately to errors that are likely to occur. You place code that might cause an error in the `try` block. Code that should run in response to an error goes in the `except` block. Code that should run only if the `try` block was successful goes in the `else` block.

Catching an exception

```
prompt = "How many tickets do you need? "
num_tickets = input(prompt)

try:
    num_tickets = int(num_tickets)
except ValueError:
    print("Please try again.")
else:
    print("Your tickets are printing.")
```

Zen of Python

Simple is better than complex

If you have a choice between a simple and a complex solution, and both work, use the simple solution. Your code will be easier to maintain, and it will be easier for you and others to build on that code later on.

