Python Cheat Sheet

Variables and Strings

Variables act like containers that hold data and give it a name.Strings are blocks of text, created by placing characters between single or double quotation marks.Python's f-strings make it possible to insert variables right into the text, allowing for quick and readable message creation.

Hello World

```
print("hello world")
```

Hello World with a variable

```
var = "Hello world"
print(var)
```

f-strings (using variables in strings)

```
first_name = 'Nikola'
last_name = 'Tesla'
full_name = f"{first_name} {last_name}"
print(full_name)
```

Lists

A list holds multiple items in order. You can access them by index or loop through them.

Make a List

```
cars = ['Audi', 'BMW', 'Ford']
```

Get the first item in a list

```
first_car = cars[0]
```

Get the last item in a list

```
last_car = cars[-1]
```

Looping through a list

```
for car in cars:
    print(bike)
```

Adding items to a list

```
cars = []
cars.append('Nissan')
cars.append('Kia')
```

Making numerical lists

```
squares = []
for x in range(1, 8):
    squares.append(x**2)
```

List comprehensions

```
squares = [x**2 for x in range(1, 12)]
```

Slicing a list

```
food = ['Apple', 'Lemon', 'Orange', 'Kiwi']
first_two = finishers[:2]
```

Copying a list

```
copy_of_food = food[:]
```

Changing an element

```
food[0] = 'Rice'
food[2] = 'Pizza'
```

Inserting elements at a particular position

```
food.insert(1, 'Pasta')
food.insert(3, 'Beef')
```

Removing an item by its value

```
food.remove('Rice')
```

Inserting elements at a particular position

```
num_foods = len(food)
print(f"We have {num_foods} foods.")
```

Touples

Tuples are like lists, but their contents are fixed and cannot be changed.

Making a tuple

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

If statements

If statements check conditions and execute code when those conditions are met.

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

```
'Apple' in food
'Beef' in food
```

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
```

Testing if a value is not in a list

```
banned_users = ['Niko', 'Lana', 'Piksi']
user = 'Anej'

if user not in banned_users:
    print("You can play!")
```

Dictionaries

Dictionaries hold data as key-value pairs, linking each key to its associated value.

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 = {'Tanja': 3, 'Suzana': 8, 'Pia': 47}

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

Looping through all keys

```
fav_numbers = {'Tanja': 3, 'Suzana': 8, 'Pia': 47}

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

Looping through all the values

```
fav_numbers = {'Tanja': 3, 'Suzana': 8, 'Pia': 47}

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

Using a loop to make a dictionary

```
squares = {}
for x in range(5):
    squares[x] = x**2
```

Using a loop to make a dictionary

```
group_1 = ['kai', 'abe', 'ada', 'gus', 'zoe']
group_2 = ['jen', 'eva', 'dan', 'isa', 'meg']
pairings = {name:name_2}
    for name, name_2 in zip(group_1, group_2)}
```

Niko Korošec

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)
```

While loops

A while loop runs code repeatedly while a condition is true, useful when the number of repetitions is unknown.

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? ")

    if msg != 'quit':
        print(msg)
```

Using continue in a loop

```
banned_users = ['Jure', 'Fredi', 'Grega', 'Ane']

prompt = "\nAdd a player to your team."
prompt += "\nEnter 'quit' when you're done. "

players = []
while True:
    player = input(prompt)

    if player == 'quit':
        break
    elif player in banned_users:
        print(f"{player is banned!")
        continue
    else:
        players.append(player)

print("\nYour team:")
for player in players:
    print(player)
```

A simple while loop

```
while True:
    print("Loop will run forever!")
```

Functions

Functions are named code blocks for a specific task. Arguments are values you pass in; parameters are the variables that receive them.

A simple function

```
def greet_user():
    print("Hello!")

greet_user()
```

Passing an argument

```
def greet_user(username):
    print(f"Hello, {username}!")

greet_user('Jure')
```

Default values for parameters

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

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

Returning a value

```
def add_numbers(x, y):
    return x + y

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

Using None to make an argument optional

```
def describe_pet(animal, name=None):
    print(f"\nI have a {animal}.")
    if name:
        print(f"Its name is {name}.")

describe_pet('hamster', 'harry')
describe_pet('snake')
```

Exceptions

Exceptions help you respond appropriately to errors that arelikely to occur. You place code that might cause an error inthe try block. Code that should run in response to an errorgoes in the except block. Code that should run only if the tryblock 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.")
    print(f"You ordered {num_tickets} tickets.")
    total_price = num_tickets * 12
    print(f"Total cost: €{total_price}")
```

Reading from a file

To read from a file your program needs to specify thepath to the file, and then read the contents of the file. Theread_text() method returns a string containing the entirecontents of the file.

Reading an entire file at once

```
from pathlib import Path

path = Path('Scara.txt')
contents = path.read_text()
print(contents)
```

Working with a file's lines

It's often useful to work with individual lines from a file. Once thecontents of a file have been read, you can get the lines using thesplitlines() method.

```
from pathlib import Path

path = Path('Scara.txt')
contents = path.read_text()

lines = contents.splitlines()

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

Writing to a file

The write_text() method can be used to write text to a file. Be careful, this will write over the current file if it already exists. To append to a file, read the contents first and then rewrite the entire file.

Writing to a file

```
from pathlib import Path

path = Path("programming.txt")

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

Writing multiple lines to a file

```
from pathlib import Path

path = Path("programming.txt")

msg = "I love programming!"
msg += "\nI love Robots."
path.write_text(msg)
```

Using a loop to make a dictionary

```
from pathlib import Path

path = Path("programming.txt")
contents = path.read_text()

contents += "\nI love programming!"
contents += "\nI love Robots."

Python."path.write_text(contents)
```

Niko Korošec

Classes - OOP

A class is like a template that describes what something is and what it can do in programming. From that template, you create objects, which are the actual usable things in your program. Classes let you organize code by bundling data and behavior together, and with inheritance, you can build new classes from existing ones to avoid repeating yourself.

Creating and using a class

*Imagine modeling a car in code. A car has **attributes**—like its color, brand, or speed—that store information. It also has behaviors—like driving, braking, or honking—that are written as **methods**, which are functions defined inside the class.*

The Car class

```
class Robot:
    """A simple model of a robot."""

    def __init__(self, name, model, year):
        """Initialize robot attributes."""
        self.name = name
        self.model = model
        self.year = year
        self.battery_level = 100

    def recharge(self):
        """Recharge the robot's battery to full."""
        self.battery_level = 100
        print(f"{self.name} is fully charged.")

    def move(self, steps):
        """Simulate robot movement."""
        if self.battery_level <= 0:
            print(f"{self.name} has no power! Recharge first.")
        else:
            self.battery_level -= steps * 0.5  # battery drain
            print(f"{self.name} moved {steps} steps, battery
                  at{self.battery_level}%.")

    def greet(self):
        """Robot greets."""
        print(f"{self.name} says: Hello, human!")
```

Creating an instance from a class

```
my_robot = Robot('RoboMax', 'X200', 2025)
```

Accessing attribute values

```
print(my_robot.name)
print(my_robot.model)
```

Calling methods

```
my_robot.recharge()
my_robot.move(10)
```

Creating multiple instances

```
my_robot = Robot('RoboMax', 'X200', 2024)
my_old_robot = Robot('RoboMini', 'S100', 2018)
my_drone = Robot('SkyBot', 'D300', 2020)
```

Modifying attributes

You can change an attribute directly, or use methods to update it safely. Methods let you control and validate how values are modified.

Modifying an attribute directly

```
my_new_robot = Robot('RoboMax', 'X200', 2024)
my_new_robot.battery_level = 50  # directly set battery level
```

Use a method to update an attribute

```
def update_battery(self, new_level):
    """Update the battery level safely."""
    if new_level <= 100:
        self.battery_level = new_level
    else:
        print("Battery can't exceed 100%!")
```

Use a method to increment an attribute

```
def charge(self, amount):
    """Add charge to the battery."""
    if self.battery_level + amount <= 100:
        self.battery_level += amount
        print("Battery charged.")
    else:
        print("Battery can't exceed 100%!")
```

Class inheritance

A child class can inherit attributes and methods from a parent class. It can also add new features or override existing ones. To inherit, put the parent class name in parentheses when defining the child class.

The __init__() method for a child class

```
class ElectricRobot(Robot):
    """A simple model of an electric robot."""

    def __init__(self, name, model, year):
        """Initialize an electric robot."""
        super().__init__(name, model, year)
        self.battery_capacity = 100  # in %
        self.charge_level = 0
        self.is_operational = True
        self.tasks_completed = 0
        self.location = "dock"
```

Adding new methods to the child class

```
class ElectricRobot(Robot):
# ...init method as before...

    def charge(self):
        """Fully charge the robot."""
        self.charge_level = 100
        print("Robot is fully charged.")
```

Using child methods and parent methods

```
my_robot = ElectricRobot('Robo', 'X1', 2024)

my_robot.charge()  # Child method
my_robot.move(5)   # Parent method
```

Overriding parent methods

```
class ElectricCar(Car):
    # ...init method as before...
    def fill_tank(self):
        """Override: electric cars don't have fuel tanks."""
        print("Error: Electric cars have no fuel tank!")
```

Instances as attributes

A class can include objects of other classes as its attributes. This makes it possible for different classes to interact and work together, modeling more complex real-world entities and relationships in a program. By embedding one object inside another, you can build richer, more realistic representations of systems and their behaviors.

A powerunit class

```
class PowerUnit:
"""A power unit for an electric robot."""

    def __init__(self, capacity=85):
        self.capacity = capacity
        self.charge_level = 0

    def get_range(self):
        return 150 if self.capacity == 40 else 225 if
               self.capacity == 65 else 300
```

Using an Instance as an Attribute

```
class ElectricRobot(Robot):
    """Electric robot with a power unit."""

    def __init__(self, name, model, year):
        super().__init__(name, model, year)
        self.power_unit = PowerUnit()

    def charge(self):
        self.power_unit.charge_level = 100
        print("Robot fully charged.")
```

Using the instance

```
my_robot = ElectricRobot('Robo1', 'X200', 2024)
my_robot.charge()
print(my_robot.power_unit.get_range())
my_robot.move(5)
```

Understanding self and __init__()

- *self: Refers to the specific object created from a class. It allows attributes and methods to be accessed throughout the class. Every method in a class automatically receives self when called via an object.*
- *__init__(): A special method automatically called when a new instance is created. It initializes the object's attributes. Misspelling __init__() will prevent proper object creation.*
- *Consistency: Using self consistently ensures that each object maintains its own independent data and behavior, avoiding conflicts between instances.*

Niko Korošec