Python Cheat Sheet

Quick Reference for Top Functions and Commands

Python Syntax Basics

Comments

Comments are an important part of your code, as they allow you to explain your thought process and make your code more readable. In Python, you can create single-line comments using the hash symbol (#).

```
# This is a single-line comment.
```

For multi-line comments, you can use triple quotes (either single or double).

```
""" This is a multi-line comment. """
```

Variables

Variables in Python are used to store data. You can assign values to variables using the equals sign (=).

```
x = 5
name = "John"
```

Variable names should be descriptive and follow the naming convention of using lowercase letters and underscores for spaces.

```
user_age = 25
favorite_color = "blue"
```

Data Types

The Python language comes with several data types built-in by default. Some of the more common ones include:

- **TEXT TYPES:** str
- **BOOLEAN TYPE:** bool
- **NUMERIC TYPES:** int, float, complex
- **SEQUENCE TYPES:** list, tuple, range
- **NONE TYPE:** Nonetype

To find out the data type of any Python object, you can use the type() function. For example:

```
name = 'jane'
print(type(name))
#Output: 'str'
```

Conditional Statements

Conditional statements in Python allow you to execute different codes based on certain conditions. The common conditional statements are **'if'**, **'elif'**, and **'else'**.

```
if condition:
    # Code to execute if the condition is true
elif another_condition:
    # Code to execute if the another_condition is true
else:
    # Code to execute if none of the conditions are true
```

Loops

A loop is used to repeatedly execute a block of code. Python has two types of loops: a **'for'** loop and a **'while'** loop.

Let's take a look at both of them:

For loops:

```
for variable in iterable:
    # Code to execute for each element in the iterable
```

While loops:

```
while condition:
    # Code to execute while the condition is true
```

Inside these loops, you can use conditional and control statements to control your program's flow.

Functions

Functions in Python are blocks of code that perform specific tasks. You can define a function using the **'def'** keyword, followed by the function name and parentheses containing any input parameters.

```
def function_name(parameters):
    # Code to execute
    return result
```

To call a function, use the function name followed by parentheses containing the necessary arguments.

```
function_name(arguments)
```

Now that we've gone over the Python basics, let's move on to some more advanced topics.

Data Structures

Lists

A list in Python is a mutable, ordered sequence of elements. To create a list, use square brackets and separate the elements with commas.

Python lists can hold a variety of data types like strings, integers, booleans, etc. Here are some examples of operations you can perform with Python lists:

- Create a list:

```
my_list = [1, 2, 3]
```

- Access elements:

```
my_list[0]
```

- Add an element:

```
my_list.append(4)
```

Tuples

A tuple is similar to a list, but it is immutable, which means you cannot change its elements once created. You can create a tuple by using parentheses and separating the elements with commas.

Here are some examples of tuple operations:

- Create a tuple:

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

- Access elements:

```
my_tuple[0] #Output: 1
```

Sets

A set is an unordered collection of unique elements. You can create a set using the set() function or curly braces.

It can also hold a variety of data types, as long as they are unique. Here are some examples of set operations:

- Create a set:

```
my_set = {1, 2, 3}
```

- Add an element:

```
my_set.add(4)
```

- Remove an element:

```
my_set.remove(1)
```

Dictionaries

A dictionary is an unordered collection of key-value pairs, where the keys are unique. You can create a dictionary using curly braces and separating the keys and values with colons. Here are some examples of dictionary operations:

- Create a dictionary:

```
my_dict = {'key1': 'value1', 'key2': 'value2'}
```

- Access elements:

```
my_dict['key1'] #Output:'value1'
```

- Add a key-value pair:

```
my_dict['key3'] = 'value3'
```

- Remove a key-value pair:

```
del my_dict['key1']
```

Remember to practice and explore these data structures in your Python projects to become more proficient in their usage.

File I/O

Reading Files

To read a file, you first need to open it using the built-in **open()** function, with the mode parameter set to **'r'** for reading:

```
file_obj = open('file_path', 'r')
```

Now that your file is open, you can use different methods to read its content:

- **read():** Reads the entire content of the file.
- **readline():** Reads a single line from the file.
- **readlines():** Returns a list of all lines in the file.

It's important to remember to close the file once you've finished working with it:

```
file_obj.close()
```

Alternatively, you can use the with statement, which automatically closes the file after the block of code completes:

```
with open('file_path', 'r') as file_obj:
    content = file_obj.read()
```

Writing Files

To create a new file or overwrite an existing one, open the file with mode **'w'**:

```
file_obj = open('file_path', 'w')
```

Write data to the file using the **write()** method:

```
file_obj.write('This is a line of text.')
```

Don't forget to close the file:

```
file_obj.close()
```

Again, consider using the with statement for a more concise and safer way to handle files:

```
with open('file_path', 'w') as file_obj:
    file_obj.write('This is a line of text.')
```

Appending To Files

To add content to an existing file without overwriting it, open the file with mode **'a'**:

```
file_obj = open('file_path', 'a')
```

Use the **write()** method to append data to the file:

```
file_obj.write('This is an extra line of text.')
```

And, as always, close the file when you're done:

```
file_obj.close()
```

For a more efficient and cleaner approach, use the with statement:

```
with open('file_path', 'a') as file_obj:
    file_obj.write('This is an extra line of text.')
```

By following these steps and examples, you can efficiently navigate file operations in your Python applications. Remember to always close your files after working with them to avoid potential issues and resource leaks.

Error Handling

Try And Except

To handle exceptions in your code, you can use the try and except blocks. The **try** block contains the code that might raise an error, whereas the except block helps you handle that exception, ensuring your program continues running smoothly.

Here's an example:

```
try:
    quotient = 5 / 0
except ZeroDivisionError as e:
    print("Oops! You're trying to divide by zero.")
```

In this case, the code inside the try block will raise a ZeroDivisionError exception. Since we have an except block to handle this specific exception, it will catch the error and print the message to alert you about the issue.

Finally

The **finally** block is used when you want to ensure that a specific block of code is executed, no matter the outcome of the try and except blocks. This is especially useful for releasing resources or closing files or connections, even if an exception occurs, ensuring a clean exit.

Here's an example:

```
try:
    # Your code here
except ZeroDivisionError as e:
    # Exception handling
finally:
    print("This will run no matter the outcome of the
    try and except blocks.")
```

Raising Exceptions

You can also raise custom exceptions in your code to trigger error handling when specific conditions are met. To do this, you can use the **raise** statement followed by the exception you want to raise (either built-in or custom exception).

For instance:

```
def validate_age(age):
    if age < 0:
        raise ValueError("Age cannot be a negative value.")
try:
    validate_age(-3)
except ValueError as ve:
    print(ve)
```

In this example, we've defined a custom function to validate an age value. If the provided age is less than zero, we raise a **ValueError** with a custom message. When calling this function, you should wrap it in a try-except block to handle the exception properly.

Modules And Packages

Importing Modules

Modules in Python are files containing reusable code, such as functions, classes, or variables. Python offers several modules and packages for different tasks like data science, machine learning, robotics, etc.

To use a module's contents in your code, you need to import it first. Here are a few different ways to import a module:

- **import <module_name>:** This imports the entire module, and you can access its contents using the syntax **'module_name.content_name.'**

  For example:

  ```
  import random
  c = random.ranint()
  ```

- **from <module_name> import <content_name>:**

  This imports a specific content (function or variable) from the module, and you can use it directly without referencing the module name.

  ```
  from math import sin
  c = sin(1.57)
  ```

- **from <module_name> import *:** This imports all contents of the module. Be careful with this method as it can lead to conflicts if different modules have contents with the same name.

Some commonly used built-in Python modules include:

1. **math:** Provides mathematical functions and constants
2. **random:** Generates random numbers and provides related functions
3. **datetime:** Handles date and time operations
4. **os:** Interacts with the operating system and manages files and directories

Creating Packages

Packages in Python are collections of related modules. They help you organize your code into logical and functional units. To create a package:

1. Create a new directory with the desired package name.
2. Add an empty file named **init.py** to the directory. This file indicates to Python that the directory should be treated as a package.
3. Add your module files (with the .py extension) to the directory.

Now, you can import the package or its modules into your Python scripts. To import a module from a package, use the syntax:

```
import <package_name.module_name>
```

Structure your code with modules and packages to make it more organized and maintainable. This will also make it easier for you and others to navigate and comprehend your codebase.

Object-Oriented Programming

Classes

A class is a blueprint for creating objects. It defines the data (attributes) and functionality (methods) of the objects. To begin creating your own class, use the **"class"** keyword followed by the class name:

```
class ClassName:
    # Class attributes and methods
```

To add attributes and methods, simply define them within the class block. For example:

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed
    def bark(self):
        print("Woof!")
```

In this example, a new Dog object can be created with a name and breed, and it has a bark method that prints **"Woof!"** when called.

Inheritance

Inheritance allows one class to inherit attributes and methods from another class, enabling code reusability and modularity. The class that inherits is called a subclass or derived class, while the class being inherited from is called the base class or superclass.

To implement inheritance, add the name of the superclass in parentheses after the subclass name:

```
class SubclassName(SuperclassName):
    # Subclass attributes and methods
```

For instance, you could create a subclass "Poodle" from a "Dog" class:

```
class Poodle(Dog):
    def show_trick(self):
        print("The poodle does a trick.")
```

A Poodle object would now have all the attributes and methods of the Dog class, as well as its own show_trick method.

Encapsulation

Encapsulation is the practice of wrapping data and methods that operate on that data within a single unit, an object in this case. This promotes a clear separation between an object's internal implementation and its external interface.

Python employs name mangling to achieve encapsulation for class members by adding a double underscore prefix to the attribute name, making it seemingly private.

```
class Example:
    def __init__(self):
        self.__private_attribute = "I'm private!"
    def __private_method(self):
        print("You can't see me!")
```

Although you can still technically access these private members in Python, doing so is strongly discouraged as it violates encapsulation principles.

By understanding and implementing classes, inheritance, and encapsulation in your Python programs, you can utilize the power and flexibility of Object-Oriented Programming to create clean, modular, and reusable code.

Helpful Python Libraries

NumPy

NumPy is a popular Python library for mathematical and scientific computing. With its powerful N-dimensional array object, you can handle a wide range of mathematical operations, such as:

- Basic mathematical functions
- Linear algebra
- Fourier analysis
- Random number generation

NumPy's efficient array manipulations make it particularly suitable for projects that require numerical calculations.

Pandas

Pandas is a powerful data analysis and manipulation library that you can use to work with structured data. It's also very popular in the data science community due to the wide array of tools it provides for handling data.

Some of its features include:

- Data structures like Series (1D) and DataFrame (2D)
- Data cleaning and preparation
- Statistical analysis
- Time series functionality

By utilizing Pandas, you can easily import, analyze, and manipulate data in a variety of formats, such as CSV, Excel, and SQL databases. If you're interested in Pandas, you can check out our video on **How To Resample Time Series Data Using Pandas To Enhance Analysis:**

Youtube Reference

Requests

The Requests library simplifies the process of handling HTTP requests in Python. With this library, you can easily send and receive HTTP requests, such as GET, POST, and DELETE. Some key features include:

- Handling redirects and following links on web pages
- Adding headers, form data, and query parameters via simple Python libraries
- Managing cookies and sessions

Using Requests, you can quickly and efficiently interact with various web services and APIs.

Beautiful Soup

Beautiful Soup is a Python library for web scraping, which allows you to extract data from HTML and XML documents. Some of its key features include:

- Searching for specific tags or CSS classes
- Navigating and modifying parsed trees
- Extracting relevant information based on tag attributes

By using Beautiful Soup in conjunction with Requests, you can create powerful web scraping applications that gather information from a wide array of websites.