Rapt ‘N Rel

Python Documentation

Welcome to the world of Python programming! Python is a popular, high-level programming language that is widely used in various fields, including web development, data analysis, artificial intelligence, and scientific computing. In this tutorial textbook, we will introduce you to Python programming and provide you with a solid foundation to start your journey as a Python developer.

This tutorial is designed for beginners who have no prior experience in programming or Python. We will start by explaining the basics of programming, including variables, data types, and control flow. We will then move on to more advanced topics, such as functions, modules, and object-oriented programming.

By the end of this tutorial, you will have a solid understanding of Python programming and be ready to take on more advanced topics. Whether you want to build web applications, analyze data, or build artificial intelligence applications, Python is a versatile and powerful tool that can help you achieve your goals.

So, let's get started and explore the exciting world of Python programming together!

Author: *Emmanuel Oluwafunso Oladosu*

Rapt ‘N Rel

Python Documentation

TABLE OF CONTENT

- Package

File Handling in Python

- Operations
- Directory
- Exception

Object Oriented Programming in Python

- Classes and Objects
- Inheritance
- Operator Overloading

Advanced Concepts in Python

- Pythonic Features
- Pythonic Features II

Date & Time

- Python datetime Module
- Python time Module

Practice

- Easy
- Medium
- Difficult
- Very Difficult

Interview Questions

- Common
- Tough
- Missed

Rapt ‘N Rel

Python Documentation

INTRODUCTION TO PYTHON

What Is Python?

Python is a high-level, interpreted programming language that is widely used for both web development and scientific computing. It was first released in 1991 by Guido van Rossum and has since become one of the most popular programming languages in the world.

Python has a simple and easy-to-learn syntax, which makes it an ideal language for beginners to start programming. It uses indentation to indicate the block of code, which makes the code more readable and easier to understand.

Python is also known for its powerful standard library, which provides a wide range of modules and packages for tasks such as file I/O, network programming, and regular expressions. Additionally, there are many third-party libraries available that can be easily installed and used in Python.

Some of the key features of Python include:

- Object-oriented programming: Python supports object-oriented programming, which allows you to define classes and objects with their own attributes and methods.
- Dynamic typing: Python is dynamically typed, which means you don't have to declare the data type of a variable before using it.
- Interpreted: Python is an interpreted language, which means the code is executed line by line rather than being compiled into machine code.
- Cross-platform: Python code can run on multiple platforms such as Windows, macOS, and Linux.
- Extensive standard library: Python comes with a large standard library that provides many useful modules and packages.
- Large community: Python has a large and active community of developers who contribute to the language, create new libraries, and provide support to other developers.

Rapt ‘N Rel

Python Documentation

Python is widely used in various fields such as web development, data science, machine learning, and artificial intelligence. It is also used in software development, game development, and automation.

Some popular web frameworks built on top of Python include Django, Flask, and Pyramid. In the data science and machine learning domain, Python has become the go-to language for tasks such as data analysis, data visualization, and machine learning model development. Popular libraries for these tasks include NumPy, Pandas, Matplotlib, and Scikit-learn.

Overall, Python is a versatile and powerful programming language that is easy to learn and widely used.

Setting Up

Python is an easy-to-learn and widely used programming language.
In order to get started with Python you will need to:

- Install Python: First, you need to install Python on your computer.
- Install an Integrated Development Environment (IDE): An IDE is a software application that provides comprehensive facilities for software development. You can choose any IDE of your choice; some popular ones include PyCharm, Spyder, and IDLE.

In this lesson we will be using PyCharm.

Here's a brief guide to get you started with Python:

- Install Python: Go to the official website of Python, https://www.python.org/downloads/ and download the latest version of Python suitable for your operating system. Follow the instructions to complete the installation.
- Install PyCharm: Go to the official website of PyCharm, https://www.jetbrains.com/pycharm/download/ and download the Community Edition of PyCharm. Follow the instructions to complete the installation.
- Create a new project: Open PyCharm and click on "Create New Project". Give your project a name and choose a location on your computer where you want to save it.

- Choose Python interpreter: Choose the Python interpreter you just installed in the first step. If you installed Python using default settings, the interpreter should be located in the following directory: "C:\Program Files\Python<version>".

- Write your code: Once you have created a project, you can start writing your Python code in the editor provided by PyCharm.

- Run your code: You can run your Python code directly from PyCharm by clicking on the "Run" button or by pressing a shortcut key. The output of your program will be displayed in the console or output window of PyCharm.

- Configure PyCharm: You can customize PyCharm's settings to suit your needs. For example, you can change the color scheme, font size, and keyboard shortcuts.

That's it! You are now ready to start coding in Python using PyCharm.

The Basics

Now that you’re all set, let’s type your first python programme:

```
print("Greetings Earth!")
```

Great!

Syntax

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.

- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.

- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Rapt ‘N Rel

Python Documentation

Variables

In programming, a variable is a reserved memory location that stores and represents data values. Here is an example:

```
numberOfCourses = 8
```

Here, numberOfCourses is the variable storing the value 8.

As demonstrated in the preceding example, the assignment operator (=) is used to assign a value to a variable.

```
# assign value to full_name variable
full_name = "Shepherd Umanah"
print(full_name)
# Output: Shepherd Umanah
```

In the aforementioned example, we assigned the value 'Shepherd Umanah' to the variable full_name. We subsequently printed the value of the name variable.

NB: Python is a dynamically-typed language, which means that you do not need to explicitly specify the variable type. The interpreter automatically infers that 'Shepherd Umanah' is a string and declares the full_name variable as a string type.

Changing the Value of a Variable:

```
full_name = 'Shepherd Umanah'
print(full_name)
# assigning a new value to site_name
full_name = 'Abiola Ojo'
print(full_name)
```

Assigning Multiple Values to Multiple Variables:

```
a, b, c = 5, 4.5, 'CGPA'
print(a)  # prints 5
print(b)  # prints 4.5
```

```
print(c)   # prints CGPA
```

Assigning the Same Value to Multiple Variables at Once:

```
course1 = course2  = 'python'
print(site1)   # prints python
print(site2)   # prints python
```

Comments

In computer programming, comments serve as annotations that enhance the readability and understanding of code. Comments are not executed by the interpreter and are intended for human readers, particularly other programmers.

For instance,

```
# declare and initialize a variable
a = 25
# print the output
print(a)
```

The following comments have been utilized in this code snippet:

- Declare and initialize a variable
- Print the output

Keywords

In Python programming, keywords are words that are predefined and reserved by the compiler, and have special meanings. They define the syntax and structure of the language, and cannot be used as variable names, function names, or any other identifier. The keywords must be written in lowercase, except for True, False, and None, and must be used as they are. Below is a list of all the keywords.

Rapt ‘N Rel

Python Documentation

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Identifiers are the name given to variables, classes, methods, etc. For example,

```
website = 'Rapt ‘N Rel'
```

Here, website is a variable (an identifier) which holds the value 'Rapt ‘N Rel'.

We cannot use keywords as variable names as they are reserved names that are built-in to Python. For example,

```
except = 'Rapt ‘N Rel'
```

The above code is **wrong** because we have used except as a variable name.

Rapt ‘N Rel

Python Documentation

The Guidelines for Naming an Identifier Are as Follows:

- Identifiers cannot be a reserved keyword.
- Identifiers are case-sensitive.
- They may consist of a combination of letters and digits, but must begin with a letter or underscore (_). The first character of an identifier cannot be a digit.
- It is recommended to start an identifier with a letter rather than an underscore.
- Spaces are not allowed in identifiers.
- Special symbols like !, @, #, $, etc. cannot be used in identifiers.

Valid Identifiers	Invalid Identifiers
learn	le@rn
returnValue	return
highest_grade	highest grade
python1	1python
convertTo_string	convert to_string

Python is a case-sensitive language. This means, Learning and learning are not the same.

Always give the identifiers a name that makes sense. While c = 3 is a valid name, writing count = 3 would make more sense, and it would be easier to figure out what it represents when you look at your code after a long gap.

Multiple words can be separated using an underscore, like rapt_n_rel .

The Basics II

This section will provide an overview of Python constants and literals, along with taking input and giving output.

Constants

A constant is a specific type of variable whose value remains fixed and cannot be modified. In Python, constants are typically declared and assigned within a module (a distinct file that contains variables, functions, and so on that are imported into the primary file).

Here is an example of how to declare constants in a separate file and subsequently use them in the main file:

Create a constant.py:

```
# declare constants
PI = 3.14159265
GRAVITY = 9.8
```

Create a main.py:

```
# import constant file we created above
import constant
print(constant.PI) # prints 3.14159265
print(constant.GRAVITY) # prints 9.8
```

In the previous illustration, we generated the constant.py module file, and set the constant values for PI and GRAVITY.

Subsequently, we created the main.py file and imported the constant module. Lastly, we printed out the values of the constants.

NB: In actuality, constants are not employed in Python. Although naming constants in all capital letters is a convention to distinguish them from variables, it does not actually prevent reassignment.

Literals

In a program, literals are representations of fixed values. These values can include numbers, characters, strings, and so on, such as 'Hello, World!', 12, 23.0, 'C', etc.

Literals are frequently used to assign values to variables or constants, as shown in this example:

```
name = 'Sharon'
```

The expression mentioned above contains a variable named "name" and a string literal "Sharon".

Types

- Numeric
  - Decimal (e.g. 8, 20)
  - Binary (e.g. 0b1001, 0b111)
  - Octal (e.g. 0o46)
  - Hexadecimal (e.g. 0x46)
  - Floating (e.g. 8.78, 2.5)
  - Complex (e.g. 8+3j)

- Boolean
  - True
  - False

- String (e.g ‘Emmanuella’)

- Special
  - None (to specify a null variable)

Rapt ‘N Rel

Python Documentation

Literal Collection

Python has four types of literal collections, namely, list literals, tuple literals, dictionary literals, and set literals. These collections allow you to store multiple values of different types in a single variable. Lists and dictionaries are mutable, meaning that you can add, remove, or modify their elements, while tuples and sets are immutable, meaning that their values cannot be changed once they are defined.

```
# list literal
developers = ["Ella", "Dosu", "Sharon"]
print(developers)

# tuple literal
numbers = (1, 2, 3)
print(numbers)

# dictionary literal
careerPaths = {'dataScience':'python',
'webDev':'javascript', 'android':'kotlin'}
print(careerPaths)

# set literal
vowels= {'a', 'e', 'i' , 'o', 'u'}
print(vowels)
```

I/O

This tutorial aims to provide easy-to-follow examples on how to display output to users and receive input from users in Python.

Output

```
print('Learning made fascinating!')
# Output: Learning made fascinating!
```

Print() Syntax:

```
print(*objects, sep=' ', end='\n', file=sys.stdout,
flush=False)
```

- objects is the item(s) to be printed. It can be one or more objects separated by commas.
- sep is the separator between the objects. The default value is a space character.
- end is the character added to the end of the output. The default value is a newline character.
- file is the output stream where the printed text will be sent. The default value is sys.stdout, which represents the console output.
- flush is a Boolean value that determines if the output stream should be flushed (True) or not (False) after printing. The default value is False.

Printing multiple objects with a separator:

```
x = 10
y = 20
print("The value of x is", x, "and the value of y
is", y, ".")
#The value of x is 10 and the value of y is 20.
```

Changing the separator:

```
name = "Emmanuel"
age = 30
print("My name is", name, "and I am", age, "years
old.", sep="***")
```

Rapt ‘N Rel

Python Documentation

```
# My name is***Emmanuel***and I am***30***years old.
```

Printing without a newline:

```
print("Hello,", end=" ")
print("world!")
# Hello, world!
```

Writing to a file:

```
with open("output.txt", "w") as f:
    print("This text is written to a file.", file=f)
```

This will create a file named output.txt in the current directory and write the specified text to it.

Printing Python Variables and Literals:

```
# Printing variables
x = 5
y = "hello"
print(x)   # prints the value of x, which is 5
print(y)   # prints the value of y, which is "hello"

# Printing literals
print("This is a string literal.")   # prints the
string "This is a string literal."
print(42)   # prints the integer 42
print(3.14)   # prints the float 3.14
print(True)   # prints the boolean value True

# Printing multiple values at once
name = "Alice"
```

```
age = 25
print("My name is", name, "and I am", age, "years
old.")   # prints "My name is Alice and I am 25 years
old."
```

Concatenation

```
first_name = "Emmanuel"
last_name = "Oladosu"
full_name = first_name + " " + last_name   #
concatenating the first and last name with a space in
between
print("My name is " + full_name + ".")   # printing
the concatenated string
# Output: My name is Emmanuel Oladosu.
```

In this example, we first define two variables first_name and last_name which contain the strings "Emmanuel" and "Oladosu" respectively. We then concatenate these strings with a space in between and store the result in a new variable full_name. Finally, we print the concatenated string using the print() function and the + operator to concatenate the string "My name is " with the full_name variable.

Output Formatting

Output formatting in Python refers to the process of displaying data in a specific format, such as aligning columns, adding padding, and so on. Python provides several ways to format the output of a program, including:

Using the % operator:

This operator can be used to format strings using placeholders for values. For example:

```
name = "Emmanuel"
```

```
age = 25
print("My name is %s and I am %d years old." % (name,
age))
```

In this example, %s is a placeholder for a string, and %d is a placeholder for an integer. The values for these placeholders are provided in a tuple after the % operator.

Using the str.format() method:

This method can be used to format strings using curly braces {} as placeholders. For example:

```
name = "Emmanuel"
age = 25
print("My name is {} and I am {} years
old.".format(name, age))
```

In this example, {} is a placeholder that will be replaced with the values provided in the format() method.

Using f-strings:

This is a newer way to format strings in Python, introduced in Python 3.6. It allows you to embed expressions inside string literals, using the syntax f"...". For example:

```
name = "Emmanuel"
age = 25
print(f"My name is {name} and I am {age} years old.")
```

In this example, the expressions inside the curly braces {} will be evaluated and the result will be included in the string.

All of these methods allow you to format the output of a program in different ways, depending on your needs.

Rapt ‘N Rel

Python Documentation

Input

In Python, the input() function is used to take input from the user. It allows the user to enter data through the keyboard, which can then be used by the program for further processing. Here's an example of how to use the input() function:

```
name = input("Please enter your name: ")
print("Hello, " + name + "!")
```

In this example, the input() function is used to prompt the user to enter their name. The text "Please enter your name: " is displayed on the screen, and the program waits for the user to enter their name. Once the user enters their name and presses enter, the input is stored in the name variable. The program then prints a message that includes the user's name using string concatenation.

It's important to note that the input() function always returns a string, even if the user enters a number. If you need to convert the input to a different data type, you can use functions like int() or float(). Here's an example:

```
age = input("Please enter your age: ")
age = int(age)    # convert the input to an integer
print("Next year, you will be " + str(age + 1) + "
years old.")
```

In this example, the input() function is used to prompt the user to enter their age. The input is then converted to an integer using the int() function, and stored in the age variable. The program then adds 1 to the age and prints a message that includes the new age. Note that we have to use the str() function to convert the integer back to a string before concatenating it with other strings.

Converting user input into a number:

You can also convert user input into a number in Python by wrapping the input() function inside the int() function or the float() function. This is a common shorthand used in Python. Here's an example:

```
number_int = int(input("Please enter an integer: "))
number_float = float(input("Please enter a float: "))
```

```
print("You entered the number", number_int, "which is an integer.")
print("You entered the number", number_float, "which is a float.")
```

Primitive Data Types

Data types are used in computer programming to specify the kind of data that can be stored in a variable. For instance,

```
num = 24
```

Here, 24 (an integer) is assigned to the num variable. So the data type of num is of the int class.

Integers and Float

In Python, the number category includes both integers and floating-point numbers, which are represented by the int and float classes, respectively.

The int class is designed to contain signed integers of unlimited length. The float class holds floating decimal points and it's accurate up to 15 decimal places.

To determine the class of a variable or value, the type() function can be utilized.

For instance:

```
num1 = 2
print(num1, 'is of type', type(num1))
# Output: 2 is of type <class 'int'>

num2 = 4.5
print(num2, 'is of type', type(num2))
```

```
# Output: 4.5 is of type <class 'float'>
```

Considering the above example:

- The value 2 is an integer, thus, the type() function returns int as the class of num1, i.e., <class 'int'>.
- The value 4.5 is a floating-point number, therefore, type() returns float as the class of num2, i.e., <class 'float'>.

Strings

A string is a collection of characters that can be represented using either single or double quotes.

For instance,

```
name = 'Dosu'
print(name)

message = 'Founder of Nuevicsu Tech'
print(message)
```

OPERATORS IN PYTHON

In Python, operators are symbols that represent a specific operation to be performed on one or more operands (values or variables). The result of the operation is returned as the output of the expression in which the operator is used.

There are several types of operators in Python, including:

- Arithmetic Operators: These operators are used to perform mathematical operations like addition, subtraction, multiplication, division, and modulus.

- Comparison Operators: These operators are used to compare two values or variables and return a Boolean value (True or False) based on the comparison.
- Logical Operators: These operators are used to perform logical operations like AND, OR, and NOT on Boolean values.
- Assignment Operators: These operators are used to assign values to variables or update the value of a variable.
- Bitwise Operators: These operators are used to perform bitwise operations on binary values.
- Membership Operators: These operators are used to test if a value or variable is a member of a sequence or collection.
- Identity Operators: These operators are used to compare the identity of two objects or variables.

Each operator has its own syntax and rules for use. Knowing how to use operators is essential to writing efficient and effective Python code.

Arithmetic Operators

They are used in Python to perform mathematical operations on numerical data types such as integers and floating-point numbers. There are five Arithmetic Operators in Python which are as follows:

Addition (+)

The addition operator is used to add two values. For example, if we want to add two numbers, say 5 and 7, we can use the + operator as follows:

```
result = 5 + 7
print(result)
# Output: 12
```

Subtraction (-)

The subtraction operator is used to subtract one value from another. For example, if we want to subtract 7 from 10, we can use the - operator as follows:

```
result = 10 - 7
print(result)
# Output: 3
```

Multiplication (*)

The multiplication operator is used to multiply two values. For example, if we want to multiply 5 and 7, we can use the * operator as follows:

```
result = 5 * 7
print(result)
# Output: 35
```

Division (/)

The division operator is used to divide one value by another. For example, if we want to divide 10 by 2, we can use the / operator as follows:

```
result = 10 / 2
print(result)
# Output: 5.0 (Note that the output is a float
because one of the operands is a float.)
```

Modulus (%)

The modulus operator is used to get the remainder of a division operation. For example, if we want to get the remainder when 10 is divided by 3, we can use the % operator as follows:

```
result = 10 % 3
print(result)
# Output: 1
```

Comparison Operators

Comparison Operators are used to compare two values or variables and return a Boolean value (True or False) based on the comparison. There are six Comparison Operators in Python which are as follows:

Equal to (==)

The equal to operator is used to check if two values are equal. For example, if we want to check if the value of x is equal to 5, we can use the == operator as follows:

```
x = 5
print(x == 5)
# Output: True
```

Not equal to (!=)

The not equal to operator is used to check if two values are not equal. For example, if we want to check if the value of x is not equal to 5, we can use the != operator as follows:

```
x = 6
print(x != 5)
# Output: True
```

Greater than (>)

The greater than operator is used to check if one value is greater than another value. For example, if we want to check if the value of x is greater than 5, we can use the > operator as follows:

```
x = 6
print(x > 5)
# Output: True
```

Less than (<)

The less than operator is used to check if one value is less than another value. For example, if we want to check if the value of x is less than 5, we can use the < operator as follows:

```
x = 4
print(x < 5)
# Output: True
```

Greater than or equal to (>=)

The greater than or equal to operator is used to check if one value is greater than or equal to another value. For example, if we want to check if the value of x is greater than or equal to 5, we can use the >= operator as follows:

```
x = 6
print(x >= 5)
# Output: True
```

Less than or equal to (<=)

The less than or equal to operator is used to check if one value is less than or equal to another value. For example, if we want to check if the value of x is less than or equal to 5, we can use the <= operator as follows:

```
x = 4
print(x <= 5)
# Output: True
```

Assignment Operators

Assignment operators in Python are used to assign values to variables. They are a shorthand way of writing expressions that combine an assignment operation with another operation, such as arithmetic or bitwise operations.

Here are some examples of assignment operators in Python:

```
a = 5  # Simple assignment

# Using addition assignment operator
a += 3  # Equivalent to: a = a + 3

# Using subtraction assignment operator
a = 5
a -= 2   # Equivalent to: a = a - 2

# Using multiplication assignment
a *= 2  # Equivalent to: a = a * 2

# Using division assignment operator
a = 10
a /= 3   # Equivalent to: a = a / 3
```

Rapt ‘N Rel

Python Documentation

```
# Using modulus assignment operator
a = 10
a %= 3   # Equivalent to: a = a % 3

# Using exponentiation assignment operator
a = 2
a **= 3   # Equivalent to: a = a ** 3
print(a)  # Output: 8

# Using floor division assignment operator
a = 10
a //= 3   # Equivalent to: a = a // 3

# Using bitwise AND assignment operator
a = 0b1010
a &= 0b1100   # Equivalent to: a = a & 0b1100
print(bin(a)) # Output: 0b1000

# Using bitwise OR assignment operator
a = 0b1010
a |= 0b1100   # Equivalent to: a = a | 0b1100
print(bin(a)) # Output: 0b1110

# Using bitwise XOR assignment operator
a = 0b1010
a ^= 0b1100   # Equivalent to: a = a ^ 0b110
```

Rapt ‘N Rel

Python Documentation

Assignment operators are useful when we want to update the value of a variable using a shorthand notation instead of writing the full expression. They are commonly used in loops, conditional statements, and other programming constructs where the same variable is updated multiple times.

Logical Operators

They are used to perform logical operations on Boolean values (True or False). There are three Logical Operators in Python which are as follows:

AND

The AND operator returns True if both operands are True, and False otherwise. For example, if we want to check if x is greater than 0 AND less than 10, we can use the AND operator as follows:

```
x = 5
print(x > 0 and x < 10)
# Output: True
```

OR

The OR operator returns True if at least one of the operands is True, and False otherwise. For example, if we want to check if x is greater than 0 OR less than 2, we can use the OR operator as follows:

```
x = 1
print(x > 0 or x < 2)
# Output: True
```

NOT

The NOT operator returns the opposite Boolean value of the operand. If the operand is True, it returns False, and if the operand is False, it returns True. For example, if we want to check if NOT x is greater than 0, we can use the NOT operator as follows:

```
x = -1
print(not x > 0)
# Output: True
```

Logical Operators are often used in conditional statements (if/else) to control the flow of the program based on certain conditions. By using Logical Operators, we can combine multiple conditions and create complex Boolean expressions.

Bit-wise Operators

Bitwise AND (&)

This operator returns a 1 in each bit position where both operands have a 1.

```
a = 5  # 101 in binary
b = 3  # 011 in binary
result = a & b  # 001 in binary
print(result)  # Output: 1
```

Bitwise OR (|)

This operator returns a 1 in each bit position where at least one operand has a 1.

```
a = 5  # 101 in binary
b = 3  # 011 in binary
result = a | b  # 111 in binary
```

```
print(result)  # Output: 7
```

Bitwise XOR (^)

This operator returns a 1 in each bit position where only one operand has a 1.

```
a = 5  # 101 in binary
b = 3  # 011 in binary
result = a ^ b  # 110 in binary
print(result)  # Output: 6
```

Bitwise NOT (~)

This operator flips all the bits of the operand.

```
a = 5  # 101 in binary
result = ~a  # -6 in decimal
print(result)  # Output: -6
```

Bitwise left shift (<<)

This operator shifts the bits of the operand to the left by a specified number of positions.

```
a = 5  # 101 in binary
result = a << 2  # 10100 in binary
print(result)  # Output: 20
```

Bitwise right shift (>>)

This operator shifts the bits of the operand to the right by a specified number of positions.

```
a = 5  # 101 in binary
result = a >> 1  # 10 in binary
print(result)  # Output: 2
```

Bitwise operators are mainly used in low-level programming, hardware manipulation, and optimizing certain algorithms.

Membership Operators

Membership operators in Python are used to test if a value is a member of a sequence or not. There are two membership operators in Python:

in

This operator returns True if a value is found in the sequence, and False otherwise.

```
my_list = [1, 2, 3, 4, 5]

# Using 'in' operator
print(3 in my_list)  # Output: True
print(6 in my_list)  # Output: False
```

not in

This operator returns True if a value is not found in the sequence, and False otherwise.

Here are some examples of using membership operators in Python:

```
# Using 'not in' operator
print(3 not in my_list)  # Output: False
print(6 not in my_list)  # Output: True
```

In the above examples, we have a list of integers and we are using the in and not in operators to test if certain values are present in the list or not.

Membership operators can be used with other sequences as well, such as tuples, sets, and strings. They are very useful when working with data that contains multiple values or elements, such as a list of usernames, a list of product IDs, or a set of keywords.

Identity Operators

Identity operators in Python are used to compare the memory locations of two objects. There are two identity operators in Python:

is

This operator returns True if both variables point to the same object in memory, and False otherwise.

```
a = [1, 2, 3]
b = a
c = [1, 2, 3]

# Using 'is' operator
print(a is b)   # Output: True
print(a is c)   # Output: False
```

is not

This operator returns True if both variables do not point to the same object in memory, and False otherwise.

```
# Using 'is not' operator
```

```
print(a is not b)   # Output: False
print(a is not c)   # Output: True
```

In the above examples, we have three variables a, b, and c, where a and b are pointing to the same list object, while c is pointing to a different list object with the same values. We are using the is and is not operators to compare the memory locations of these objects.

Identity operators are useful when we want to check if two variables are pointing to the same object in memory, instead of just comparing their values. They are commonly used when working with mutable objects such as lists, dictionaries, and objects.

Control Flow In Python

Control flow refers to the order in which the instructions in a program are executed. In Python, control flow can be managed using various constructs, including conditional statements, loops, and function calls. Understanding control flow is essential for writing effective and efficient programs.

- Conditional statements, such as if, elif, and else, allow you to execute different blocks of code based on specific conditions. For example, an if statement can be used to check whether a variable is equal to a specific value, and execute a block of code only if the condition is True. On the other hand, an elif statement allows you to check multiple conditions, and execute different blocks of code for each condition. Finally, an else statement can be used to execute a block of code if none of the previous conditions were True.

- Loops allow you to execute a block of code multiple times. In Python, there are two types of loops: while loops and for loops. A while loop executes a block of code as long as a specific condition is True. For example, you can use a while loop to repeatedly prompt the user for input until they enter a valid value. A for loop, on the other hand, is used to iterate over a sequence of values. For example, you can use a for loop to iterate over the elements of a list, and perform a specific operation on each element.

Overall, control flow is a crucial aspect of programming in Python.

Rapt ‘N Rel

Python Documentation

Conditionals

In Python, conditionals are used to execute different blocks of code based on the outcome of a logical test. The main conditional statements in Python are if, elif, and else.

If

The if statement is used to execute a block of code if a condition is true. For example:

```
x = 10
if x > 5:
    print("x is greater than 5")
```

In this code, the if statement checks whether x is greater than 5. Since x is equal to 10, which is greater than 5, the print() statement is executed and the message "x is greater than 5" is displayed.

Elif

The elif statement is used to test additional conditions if the previous conditions have all been false. For example:

```
x = 10
if x > 20:
    print("x is greater than 20")
elif x > 15:
    print("x is greater than 15 but less than or
equal to 20")
else:
    print("x is less than or equal to 15")
```

In this code, the if statement checks whether x is greater than 20. Since x is equal to 10, the if condition is false, and the program proceeds to the next elif statement. This statement checks

whether x is greater than 15, which is also false. Finally, the else statement is executed, which prints the message "x is less than or equal to 15".

Else

The else statement is used to execute a block of code if none of the previous conditions have been true. For example:

```
x = 10
if x > 20:
    print("x is greater than 20")
elif x > 15:
    print("x is greater than 15 but less than or
equal to 20")
else:
    print("x is less than or equal to 15")
```

In this code, the else statement is executed because both the if and elif conditions are false.

You can also use logical operators such as and, or, and not to create more complex conditions. For example:

```
x = 10
y = 5
if x > 5 and y > 5:
    print("both x and y are greater than 5")
elif x > 5 or y > 5:
    print("either x or y is greater than 5")
else:
    print("neither x nor y is greater than 5")
```

In this code, the if statement checks whether both x and y are greater than 5, which is false. The elif statement checks whether either x or y is greater than 5, which is true for x. Therefore, the message "either x or y is greater than 5" is displayed.

If...else

The if...else statement is used when you want to execute one code block if a condition is true, and another code block if the condition is false. Here's an example:

```
x = 5
if x > 10:
    print("x is greater than 10")
else:
    print("x is less than or equal to 10")
```

In this example, the if statement checks whether x is greater than 10. If it is, the message "x is greater than 10" is printed. If not, the else statement is executed and the message "x is less than or equal to 10" is printed.

The if...else statement is useful when you need to execute different code based on a single condition. However, if you need to test multiple conditions, you may want to use the if...elif...else statement instead.

Here's an example of using if...else within a loop to filter a list of numbers:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = []
odd_numbers = []
for num in numbers:
    if num % 2 == 0:
        even_numbers.append(num)
    else:
        odd_numbers.append(num)
```

```
print("Even numbers:", even_numbers)
print("Odd numbers:", odd_numbers)
```

In this example, the if statement checks whether the current number in the loop is even (i.e., divisible by 2 with no remainder). If it is, the number is added to the even_numbers list. If not, the else statement is executed and the number is added to the odd_numbers list.

If...elif...else

The if...elif...else statement allows you to test multiple conditions and execute different code blocks depending on the outcome of the tests. Here's an example:

```
x = 5
if x > 10:
    print("x is greater than 10")
elif x > 5:
    print("x is greater than 5 but less than or equal to 10")
else:
    print("x is less than or equal to 5")
```

In this example, the if statement checks whether x is greater than 10. If it is, the message "x is greater than 10" is printed. If not, the elif statement checks whether x is greater than 5. If it is, the message "x is greater than 5 but less than or equal to 10" is printed. Finally, if both the if and elif statements are false, the else statement is executed and the message "x is less than or equal to 5" is printed.

Nested if

Nested if statements are used when you need to test multiple conditions within the same block of code. Here's an example:

```
x = 10
y = 5
if x > 5:
    if y > 5:
        print("both x and y are greater than 5")
    else:
        print("x is greater than 5 but y is less than
or equal to 5")
else:
    print("x is less than or equal to 5")
```

In this example, the outer if statement checks whether x is greater than 5. If it is, the inner if statement checks whether y is greater than 5. If it is, the message "both x and y are greater than 5" is printed. If not, the message "x is greater than 5 but y is less than or equal to 5" is printed. Finally, if the outer if statement is false, the else statement is executed and the message "x is less than or equal to 5" is printed.

Nested if statements can be useful when you need to test multiple conditions that are related to each other, or when you need to perform more complex logic within a single code block.

Overall, conditionals are an important part of Python programming and are used to control the flow of execution based on logical tests. By using if, elif, and else statements, along with logical operators, you can create powerful programs that make decisions based on input data and other factors.

Loops

In Python, loops are used to execute a block of code repeatedly until a specific condition is met. There are two main types of loops in Python: for loops and while loops.

Rapt ‘N Rel

Python Documentation

For Loop

A for loop is used to iterate over a sequence of items, such as a list or a string. Here's the general syntax of a for loop:

```
for item in sequence:
    # Do something with item
```

In this syntax, item is a variable that represents the current item in the sequence, and sequence is the sequence of items that you want to iterate over. The loop body consists of the instructions that you want to execute for each item in the sequence.

Here's an example of using a for loop to print each item in a list:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

In this example, the loop iterates over the list of fruits and prints each fruit in turn.

For in range() Loop

You can also use the range() function to generate a sequence of numbers to iterate over. Here's an example:

```
for i in range(1, 6):
    print(i)
```

In this example, the loop prints the numbers from 1 to 5. Note that the range() function generates a sequence of numbers up to, but not including, the second argument.

For Loop with Else

In Python, you can add an else clause to a for loop that is executed when the loop has exhausted all the items in the sequence. Here's an example:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
else:
    print("No more fruits")

# The output will be:
# apple
# banana
# cherry
# No more fruits
```

In this example, the for loop prints each fruit in the list, and the else clause is executed when the loop has completed.

If the loop is terminated by a break statement, the else clause will not be executed.

While Loop

A while loop is used to repeat a set of instructions until a specific condition is met. Here's the general syntax of a while loop:

```
while condition:
    # Do something
```

In this syntax, condition is a Boolean expression that is evaluated at the start of each iteration. The loop body consists of the instructions that you want to execute as long as the condition is true.

Here's an example of using a while loop to count down from 10 to 1:

```
i = 10
while i > 0:
```

```
    print(i)
    i -= 1
```

In this example, the loop prints the value of i and decrements it by 1 on each iteration. The loop terminates when i becomes 0.

While Loop with Else

Similar to for loops, you can also add an else clause to a while loop that is executed when the condition becomes false. Here's an example:

```
i = 1
while i <= 5:
    print(i)
    i += 1
else:
    print("Loop completed")

# The output will be:
# 1
# 2
# 3
# 4
# 5
```

In this example, the while loop prints the numbers from 1 to 5, and the else clause is executed when the condition i <= 5 becomes false. The output will be:

Loop completed
If the loop is terminated by a break statement, the else clause will not be executed.

Infinite While Loop

An infinite loop is a loop that never ends unless it is interrupted externally. An infinite loop is created by using a while loop with a condition that is always True. Here's an example:

```
while True:
    print("This is an infinite loop")
```

In this example, the while loop condition is always True, so the loop will never exit unless it is interrupted externally, such as by pressing Ctrl+C on the keyboard.

It is important to note that infinite loops should be used with caution as they can cause the program to become unresponsive or crash. Therefore, it is essential to add some form of logic or user input that can exit the loop.

For example, you can use a break statement inside the infinite loop to exit the loop based on a certain condition, like this:

```
while True:
    user_input = input("Enter 'stop' to exit the
loop: ")
    if user_input == 'stop':
        break
    print("This is an infinite loop")
```

In this example, the while loop continues indefinitely until the user enters the string 'stop'. When the user enters 'stop', the break statement is executed, and the loop is exited. Otherwise, the loop continues indefinitely, printing the message "This is an infinite loop".

Nested Loop

You can also use loops inside other loops, which is known as nested looping. This is useful when you need to iterate over a sequence of sequences, such as a 2D list or a matrix.

Here's an example of using nested loops to print the elements of a 2D list:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for row in matrix:
    for element in row:
        print(element, end=" ")
    print()
```

In this example, the outer loop iterates over each row in the matrix, and the inner loop iterates over each element in the row. The print() function is used to print each element on the same line, and the end parameter is set to a space character to separate the elements. The outer loop also includes a print() statement with no arguments to print a newline character after each row.

Overall, loops are a powerful tool for automating repetitive tasks in Python. By using loops in combination with other programming constructs like conditionals and functions, you can build complex programs that can handle a wide variety of tasks.

Break, Continue and Pass

Break Statement

The break statement is used to exit a loop prematurely. It can be used with for loops, while loops, and nested loops. When a break statement is encountered inside a loop, the loop is immediately terminated, and the program execution continues with the statement immediately following the loop.

Here's an example of a for loop that uses the break statement to exit the loop when the value of i is 3:

```
for i in range(1, 6):
    if i == 3:
        break
    print(i)

# The output will be:
```

```
# 1
# 2
```

In this example, the loop iterates through the numbers 1 to 5. When the value of i is 3, the break statement is encountered, and the loop is immediately terminated.

Continue Statement

The continue statement is used to skip the rest of the statements inside a loop for the current iteration and move to the next iteration. It can be used with for loops, while loops, and nested loops. When a continue statement is encountered inside a loop, the current iteration is immediately terminated, and the program execution continues with the next iteration of the loop.

Here's an example of a for loop that uses the continue statement to skip printing the value of i when it is 3:

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)

# The output will be:
# 1
# 2
# 4
# 5
```

In this example, the loop iterates through the numbers 1 to 5. When the value of i is 3, the continue statement is encountered, and the current iteration is skipped. The loop continues with the next iteration, printing the values of i for all other iterations.
Both break and continue statements can be used to control the flow of a loop and execute statements conditionally based on certain conditions. It's important to use them judiciously and appropriately to avoid unintended consequences and make your code more efficient.

Rapt ‘N Rel

Python Documentation

Pass Statement

The pass statement is a null operation in Python. It is used when a statement is required syntactically, but you do not want to execute any code. It is often used as a placeholder for code that has not been implemented yet, or for empty code blocks that will be filled in later.

Here's an example of a function that uses the pass statement as a placeholder for the function body:

```
def my_function():
    pass
```

In this example, the my_function function is defined with the pass statement as the function body. This means that the function does not do anything when called. You can later fill in the function body with the actual code that you want to execute.

Another use case for the pass statement is to define a class or a function that is not yet implemented. Here's an example of a class that is defined with the pass statement as the class body:

```
class MyClass:
    pass
```

In this example, the MyClass class is defined with the pass statement as the class body. This means that the class does not have any attributes or methods defined. You can later fill in the class body with the actual attributes and methods that you want to define.

The pass statement is also useful for defining empty code blocks inside loops, conditionals, and functions, where you want to defer the implementation of certain parts of the code until later. Here's an example of a for loop with an empty code block defined using the pass statement:

```
for i in range(1, 6):
    if i == 3:
        pass
    else:
        print(i)
```

```
# The output will be:
# 1
# 2
# 4
# 5
```

In this example, the loop iterates through the numbers 1 to 5. When the value of i is 3, the pass statement is encountered, and the code block inside the if statement is skipped. The loop continues with the next iteration, printing the values of i for all other iterations.

The pass statement can be used to make your code more readable and organized by providing a placeholder for future code implementation. It is a simple but powerful tool that can help you write more efficient and effective code.

Data Types In Python

Python has several built-in data types, each with its own characteristics and uses. Here is a brief description of the most commonly used data types in Python:

- Numeric Types: These are data types that represent numbers. There are three types of numeric data types in Python: integers, floating-point numbers, and complex numbers. Integers are whole numbers with no decimal point, floating-point numbers are numbers with a decimal point, and complex numbers are numbers with a real part and an imaginary part.

- Boolean Type: The boolean data type is a binary data type that represents the truth values True and False. It is used for logical operations and conditionals in Python.

- Sequence Types: These are data types that represent ordered collections of elements. There are three types of sequence data types in Python: strings, lists, and tuples. Strings are sequences of characters, lists are sequences of values that can be modified, and tuples are sequences of values that cannot be modified.

- Mapping Types: These are data types that represent key-value pairs. The most commonly used mapping data type in Python is the dictionary.

- Set Types: These are data types that represent unordered collections of unique elements. The most commonly used set data type in Python is the set.

- NoneType: The NoneType data type represents the absence of a value. It is used when a variable or function should not have a value assigned to it.

In addition to these built-in data types, Python also allows for the creation of user-defined data types using classes and objects. This allows for the creation of more complex data structures and customized data types that can be tailored to specific needs.

Numeric

Python has three built-in numeric data types:

Integers

Integers are whole numbers with no decimal point. They can be positive, negative, or zero. In Python, integers are represented by the int data type.
Here are some examples of integers in Python:

```
x = 10
y = -5
z = 0
```

Floating-point numbers

Floating-point numbers, or simply floats, are numbers with a decimal point. They can also be positive, negative, or zero. In Python, floating-point numbers are represented by the float data type.
Here are some examples of floats in Python:

```
x = 3.14
y = -2.5
z = 0.0
```

Rapt ‘N Rel

Python Documentation

Complex numbers

Complex numbers are numbers with a real part and an imaginary part. In Python, complex numbers are represented by the complex data type.
Here are some examples of complex numbers in Python:

```
x = 2 + 3j
y = -1j
```

Python also provides several built-in functions for working with numeric data types, such as abs(), round(), pow(), and divmod().

Here are some examples of how to use these functions:

```
# Absolute value of a number
x = abs(-5)
print(x)  # Output: 5

# Rounding a number
x = round(3.14159, 2)
print(x)  # Output: 3.14

# Exponentiation
x = pow(2, 3)
print(x)  # Output: 8

# Integer division and modulus
x, y = divmod(10, 3)
print(x, y)  # Output: 3 1
```

Numeric data types in Python support arithmetic operations such as addition, subtraction, multiplication, and division, as well as more complex operations such as exponentiation, floor division, and modulus.

Rapt ‘N Rel

Python Documentation

Here are some examples of arithmetic operations with numeric data types:

```
x = 10
y = 3

# Addition
print(x + y)  # Output: 13

# Subtraction
print(x - y)  # Output: 7

# Multiplication
print(x * y)  # Output: 30

# Division
print(x / y)  # Output: 3.3333333333333335

# Exponentiation
print(x ** y)  # Output: 1000

# Floor division
print(x // y)  # Output: 3

# Modulus
print(x % y)  # Output: 1
```

Overall, Python's built-in numeric data types and functions provide a robust set of tools for performing calculations and working with numbers in Python.

Rapt ‘N Rel

Python Documentation

String

In Python, a string is a sequence of characters enclosed in quotes, either single quotes (') or double quotes ("). Strings are used to represent text data in Python and are represented by the str data type.

Here are some examples of strings in Python:

```
x = "Hello, world!"
y = 'Python is fun'
z = "I'm learning Python"
```

Python strings are immutable, which means that once a string is created, it cannot be modified. However, you can create a new string by concatenating or slicing existing strings.

Here are some examples of string concatenation and slicing in Python:

```
# Concatenating strings
x = "Hello"
y = "world"
z = x + " " + y
print(z)  # Output: "Hello world"

# Slicing a string
x = "Python is fun"
print(x[0:6])  # Output: "Python"
print(x[7:])  # Output: "is fun"
```

Python also provides a wide range of built-in functions and methods for working with strings. Some of the most commonly used string methods include:

Rapt ‘N Rel

Python Documentation

- len(): returns the length of a string
- lower(): returns a copy of a string with all characters in lowercase
- upper(): returns a copy of a string with all characters in uppercase
- strip(): returns a copy of a string with leading and trailing whitespace removed
- split(): returns a list of substrings separated by a specified delimiter
- join(): returns a concatenated string of a list of strings

Here are some examples of how to use these string methods:

```
x = "Python is fun"
print(len(x))  # Output: 13

y = "HELLO, WORLD!"
print(y.lower())  # Output: "hello, world!"

z = "    Python is fun    "
print(z.strip())  # Output: "Python is fun"

a = "apple,banana,orange"
print(a.split(","))  # Output: ['apple', 'banana',
'orange']

b = ["apple", "banana", "orange"]
print(", ".join(b))  # Output: "apple, banana,
orange"
```

Overall, Python's built-in string data type and methods provide a powerful set of tools for working with text data in Python.

Rapt ‘N Rel

Python Documentation

List

A list in Python is a collection of items that are ordered and changeable. Lists are one of the most commonly used data structures in Python, and they are represented by the list data type.

Here are some examples of lists in Python:

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed_list = ["apple", 1, True, "banana"]
```

Lists in Python are mutable, which means that you can add, remove, or modify items in a list after it has been created.

Here are some examples of how to modify a list in Python:

```
# Adding items to a list
fruits = ["apple", "banana", "cherry"]
fruits.append("orange")
print(fruits)  # Output: ['apple', 'banana',
'cherry', 'orange']

# Removing items from a list
numbers = [1, 2, 3, 4, 5]
numbers.remove(3)
print(numbers)  # Output: [1, 2, 4, 5]

# Modifying items in a list
fruits = ["apple", "banana", "cherry"]
```

```
fruits[1] = "orange"
print(fruits)  # Output: ['apple', 'orange',
'cherry']
```

Python also provides a wide range of built-in functions and methods for working with lists. Some of the most commonly used list methods include:

- len(): returns the number of items in a list
- append(): adds an item to the end of a list
- remove(): removes an item from a list
- pop(): removes an item from a list and returns it
- sort(): sorts the items in a list
- reverse(): reverses the order of the items in a list
- Here are some examples of how to use these list methods:

```
fruits = ["apple", "banana", "cherry"]
print(len(fruits))  # Output: 3

numbers = [1, 5, 2, 4, 3]
numbers.sort()
print(numbers)  # Output: [1, 2, 3, 4, 5]

fruits = ["apple", "banana", "cherry"]
fruits.pop(1)
print(fruits)  # Output: ['apple', 'cherry']
```

Accessing List Items

```
languages = ["Yoruba", "French", "Arabic"]

# access element at index 0
print(languages[0])    # Yoruba

# access element at index 2
print(languages[2])    # Arabic
```

In the example provided, the index values were utilized to retrieve specific items from the "languages" list.

For instance, "languages[0]" retrieves the first item from the "languages" list, which is "Yoruba." Similarly, "languages[2]" retrieves the third item from the "languages" list, which is "Arabic."

Overall, Python's built-in list data type and methods provide a powerful set of tools for working with collections of items in Python.

Tuple

A tuple in Python is a collection of items that are ordered and immutable. Tuples are similar to lists, but once a tuple is created, you cannot modify its contents.

Here are some examples of tuples in Python:

```
fruits = ("apple", "banana", "cherry")
numbers = (1, 2, 3, 4, 5)
mixed_tuple = ("apple", 1, True, "banana")
```

Tuples are defined using parentheses () instead of square brackets [], which are used to define lists.

While you cannot modify the contents of a tuple, you can access individual items in a tuple using indexing, just like with lists. Here's an example:

```
fruits = ("apple", "banana", "cherry")
print(fruits[1])  # Output: 'banana'
```

In addition to indexing, you can also use the len() function to get the number of items in a tuple.

```
numbers = (1, 2, 3, 4, 5)
print(len(numbers))  # Output: 5
```

One of the key benefits of tuples is that they can be used as keys in Python dictionaries, while lists cannot. This is because tuples are immutable, which means that their values cannot be changed after they are created.

Python provides a number of built-in functions for working with tuples, including:

- len(): returns the number of items in a tuple
- index(): returns the index of a specified item in a tuple
- count(): returns the number of times a specified item appears in a tuple

Here are some examples of how to use these tuple methods:

```
fruits = ("apple", "banana", "cherry")
print(len(fruits))  # Output: 3

numbers = (1, 2, 3, 4, 5)
print(numbers.index(3))  # Output: 2

mixed_tuple = ("apple", 1, True, "banana", "apple")
print(mixed_tuple.count("apple"))  # Output: 2
```

Accessing Tuple Items

Here, playerNBA is a tuple with a string value Kyrie and float value 187.96.

```
playerNBA = ('Kyrie', 187.96)
# create a tuple
product = ('Microsoft', 'Xbox', 499.99)

# access element at index 0
print(product[0])    # Microsoft

# access element at index 1
print(product[1])    # Xbox
```

Overall, tuples provide a useful tool for working with collections of items in Python when you need to ensure that the contents of the collection cannot be modified after it is created.

Set

In Python, a set is an unordered collection of unique elements. The main properties of sets are:

The elements in a set are unique, meaning that there can be no duplicates.
The elements in a set are unordered, meaning that they are not stored in any particular order.
Here are some examples of sets in Python:

```
fruits = {"apple", "banana", "cherry"}
numbers = {1, 2, 3, 4, 5}
mixed_set = {"apple", 1, True, "banana"}
```

You can also create an empty set using the set() function, like this:

```
empty_set = set()
```

To add elements to a set, you can use the add() method, like this:

```
fruits = {"apple", "banana", "cherry"}
fruits.add("orange")
print(fruits)  # Output: {'apple', 'banana',
'cherry', 'orange'}
```

To remove elements from a set, you can use the remove() method, like this:

```
fruits = {"apple", "banana", "cherry"}
fruits.remove("banana")
print(fruits)  # Output: {'apple', 'cherry'}
```

You can also use the discard() method to remove an element from a set, like this:

```
fruits = {"apple", "banana", "cherry"}
fruits.discard("banana")
print(fruits)  # Output: {'apple', 'cherry'}
```

Note that if you try to remove an element that does not exist in the set using the remove() method, you will get a KeyError error. However, if you use the discard() method to remove an element that does not exist in the set, no error will be raised.

Sets also support a number of operations, such as union, intersection, and difference. Here are some examples:

```
set1 = {1, 2, 3}
set2 = {2, 3, 4}

# Union of two sets
print(set1.union(set2))  # Output: {1, 2, 3, 4}
```

```
# Intersection of two sets
print(set1.intersection(set2))  # Output: {2, 3}

# Difference of two sets
print(set1.difference(set2))  # Output: {1}
```

In addition to these basic set operations, Python also provides a number of built-in functions for working with sets, such as len(), max(), min(), and sum().

Overall, sets are a useful tool for working with collections of unique elements in Python, and they provide a number of useful operations and methods for working with these collections.

Dictionary

In Python, a dictionary is a collection of key-value pairs. The main properties of dictionaries are:

The elements in a dictionary are unordered, meaning that they are not stored in any particular order.
The keys in a dictionary are unique, meaning that they can only occur once.
The values in a dictionary can be of any data type, including other dictionaries.
Here's an example of a dictionary in Python:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
```

In this example, the keys are "name", "age", and "city", and the values are "John", 30, and "New York", respectively.

You can also create an empty dictionary using the dict() function, like this:

```
empty_dict = dict()
```

To access the values in a dictionary, you can use the key as the index, like this:

```
my_dict = {"name": "John", "age": 30, "city": "New
York"}
print(my_dict["name"])  # Output: John
```

If you try to access a key that does not exist in the dictionary, you will get a KeyError error. However, you can use the get() method to avoid this error, like this:

```
my_dict = {"name": "John", "age": 30, "city": "New
York"}
print(my_dict.get("name"))  # Output: John
print(my_dict.get("country"))  # Output: None
```

You can also add new key-value pairs to a dictionary, like this:

```
my_dict = {"name": "John", "age": 30, "city": "New
York"}
my_dict["country"] = "USA"
print(my_dict)  # Output: {'name': 'John', 'age': 30,
'city': 'New York', 'country': 'USA'}
```

To remove a key-value pair from a dictionary, you can use the pop() method, like this:

```
my_dict = {"name": "John", "age": 30, "city": "New
York"}
my_dict.pop("age")
print(my_dict)  # Output: {'name': 'John', 'city':
'New York'}
```

You can also use the del keyword to remove a key-value pair from a dictionary, like this:

```
my_dict = {"name": "John", "age": 30, "city": "New
York"}
```

```
del my_dict["age"]
print(my_dict)  # Output: {'name': 'John', 'city':
'New York'}
```

Dictionaries also support a number of operations, such as iterating over the keys, values, or key-value pairs. Here are some examples:

```
my_dict = {"name": "John", "age": 30, "city": "New
York"}

# Iterate over the keys
for key in my_dict:
    print(key)

# Iterate over the values
for value in my_dict.values():
    print(value)

# Iterate over the key-value pairs
for key, value in my_dict.items():
    print(key, value)
```

In addition to these basic dictionary operations, Python also provides a number of built-in functions for working with dictionaries, such as len(), max(), min(), and sum().

Accessing Dictionary Values

```
# create a dictionary named capital_city
capital_city = {'Oyo': 'Ibadan', 'Edo': 'Benin',
'Rivers': 'Port Harcourt'}
```

```
print(capital_city['Oyo'])  # prints Ibadan

print(capital_city['Ibadan'])  # throws error message

# create a dictionary named capital_city
capital_city = {'Oyo': 'Ibadan', 'Edo': 'Benin',
'Rivers': 'Port Harcourt'}

print(capital_city['Oyo'])  # prints Ibadan

print(capital_city['Ibadan'])  # throws error message
```

In the code provided, we have used the keys to access the corresponding values in the 'capital_city' dictionary.

For instance, since 'Nepal' is a key in the dictionary, calling 'capital_city['Oyo']' returns the value associated with it, which is 'Ibadan'.

On the other hand, calling 'capital_city['Ibadan']' is incorrect because 'Ibadan' is a value in the dictionary and not a key, hence it will return a KeyError.

Overall, dictionaries are a powerful tool for working with key-value data in Python, and they provide a number of useful operations and methods for working with this type of data.

Boolean

In Python, the boolean data type represents truth values. There are only two possible boolean values: True and False. These values are used to represent the result of a logical operation, such as a comparison or a boolean expression.

Boolean values are used extensively in Python programming for making decisions, controlling the flow of execution, and performing other logic operations.

Here are some examples of boolean expressions:

```
# Comparison operators
x = 5
y = 3
print(x > y)   # Output: True
print(x == y)  # Output: False
print(x != y)  # Output: True

# Logical operators
a = True
b = False
print(a and b)  # Output: False
print(a or b)  # Output: True
print(not a)  # Output: False
```

In Python, any value can be evaluated as a boolean value. The following values are considered False:

- False
- None
- 0 (integer)
- 0.0 (float)
- "" (empty string)
- [] (empty list)
- {} (empty dictionary)
- set() (empty set)

All other values are considered True.

You can also use the bool() function to convert a value to a boolean:

```
print(bool(0))   # Output: False
print(bool(10))   # Output: True
print(bool("hello"))   # Output: True
print(bool(""))   # Output: False
```

Boolean values can also be combined with conditional statements to control the flow of execution. Here's an example:

```
x = 5
if x > 3 and x < 10:
    print("x is between 3 and 10")
else:
    print("x is not between 3 and 10"
```

This code will print "x is between 3 and 10", since the value of x is greater than 3 and less than 10.

Overall, boolean values are a fundamental part of Python programming, and they are used extensively for making decisions and controlling the flow of execution.

Type Conversion

In Python, type conversion or type casting is the process of changing the data type of a variable from one type to another. This is an important concept in programming because different operations and functions require different data types. Python provides several built-in functions for performing type conversion, including int(), float(), str(), list(), tuple(), set(), and dict().

Rapt ‘N Rel

Python Documentation

int()

This function is used to convert a number or a string containing a number to an integer. If the argument is a string, it must contain only digits (no decimal point or sign).

Example:

```
x = 10.5
y = int(x)
print(y)     # Output: 10
```

float()

This function is used to convert a number or a string containing a number to a floating-point number.

Example:

```
x = 10
y = float(x)
print(y)     # Output: 10.0
```

str()

This function is used to convert any data type to a string.

Example:

```
x = 10
y = str(x)
print(y)     # Output: '10'
```

Rapt ‘N Rel

Python Documentation

list()

This function is used to convert any iterable (such as a tuple, set, or string) to a list.

Example:

x = (1, 2, 3)
y = list(x)
print(y) # Output: [1, 2, 3]
tuple(): This function is used to convert any iterable (such as a list, set, or string) to a tuple.

Example:

```
x = [1, 2, 3]
y = tuple(x)
print(y)     # Output: (1, 2, 3)
```

set()

This function is used to convert any iterable (such as a list or tuple) to a set.

Example:

```
x = [1, 2, 3]
y = set(x)
print(y)     # Output: {1, 2, 3}
```

dict()

This function is used to convert a sequence of key-value pairs (such as a list of tuples) to a dictionary.

Example:

```
x = [("a", 1), ("b", 2), ("c", 3)]
y = dict(x)
```

```
print(y)     # Output: {'a': 1, 'b': 2, 'c': 3}
```

It is important to note that not all types can be converted to all other types. For example, you cannot convert a string containing non-numeric characters to an integer using the int() function. In such cases, a ValueError will be raised. Similarly, you cannot convert a list containing non-hashable items to a set using the set() function.

Type conversion is often necessary when working with different data types in Python. By understanding the built-in type conversion functions, you can easily convert between different data types to perform the necessary operations and functions in your programs.

Functions In Python

Functions in Python are a set of reusable blocks of code that performs a specific task. They are defined using the keyword "def" followed by the function name and a set of parentheses containing the input parameters. The code block of a function is indented, and it can optionally return a value using the "return" keyword.

Here is the syntax of defining a function in Python:

```
def function_name(parameters):
    # function code block
    return value
```

Let's take a closer look at each part of a function:

- def: The keyword "def" indicates the start of the function definition.
- function_name: This is the name given to the function. It should be unique and descriptive of the task it performs.
- parameters: These are optional input parameters that are passed to the function when it is called. A function can have zero or more parameters separated by commas.
- code block: This is the set of instructions that the function performs. It can contain any valid Python code.

- return: This keyword is used to return a value from the function to the caller. If no value is returned, the function returns None by default.

Here is an example of a simple function that takes two numbers as input, adds them, and returns the result:

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

To call this function, we simply provide the input values as arguments:

```
sum = add_numbers(2, 3)
print(sum)  # Output: 5
```

Functions can also be called with keyword arguments, which allow us to specify the parameter values by name instead of position:

```
sum = add_numbers(x=2, y=3)
print(sum)  # Output: 5
```

Functions can also have default parameter values, which are used if no value is provided for that parameter when the function is called:

```
def add_numbers(x, y=0):
    result = x + y
    return result

sum = add_numbers(2)
print(sum)  # Output: 2

sum = add_numbers(2, 3)
print(sum)  # Output: 5
```

Rapt ‘N Rel

Python Documentation

In summary, functions in Python are a powerful feature that allows us to write reusable code that performs a specific task. By defining functions, we can make our code more modular, easier to read, and less prone to errors.

Arguments

Functions in Python can take arguments, which are inputs passed to the function when it is called. Arguments allow us to customize the behavior of a function and make it more flexible. There are four types of arguments that can be used in Python functions: positional arguments, keyword arguments, default arguments, and variable-length arguments.

Positional arguments

These are the most common types of arguments in Python functions. Positional arguments are passed to the function in the order they are defined in the function signature. Here's an example of a function that takes two positional arguments:

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

We can call this function with two positional arguments like this:

```
result = add_numbers(2, 3)
print(result)  # Output: 5
```

Keyword arguments

These allow us to pass arguments to a function by explicitly specifying the parameter names in the function call. Keyword arguments are useful when a function has many parameters or when we want to pass arguments in a different order than they are defined in the function signature. Here's an example:

```
result = add_numbers(y=3, x=2)
print(result)  # Output: 5
```

Rapt ‘N Rel

Python Documentation

Default arguments

These allow us to specify default values for parameters in a function. If a default value is provided for a parameter, it can be omitted when the function is called. Here's an example:

```
def add_numbers(x, y=0):
    return x + y

result = add_numbers(2)
print(result)  # Output: 2

result = add_numbers(2, 3)
print(result)  # Output: 5
```

In this example, the second argument "y" has a default value of 0. If we call the function with only one argument, the default value is used for "y".

Variable-length arguments

These allow us to pass a variable number of arguments to a function. There are two types of variable-length arguments in Python: *args and **kwargs.

*args

This allows us to pass a variable number of positional arguments to a function. The arguments are collected into a tuple.

```
def sum_numbers(*args):
    total = 0
    for num in args:
        total += num
```

```
    return total

result = sum_numbers(1, 2, 3)
print(result)  # Output: 6
```

**kwargs

This allows us to pass a variable number of keyword arguments to a function. The arguments are collected into a dictionary.

```
def print_values(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_values(name="John", age=30, city="New York")
# Output:
# name: John
# age: 30
# city: New York
```

In summary, Python functions can take four types of arguments: positional arguments, keyword arguments, default arguments, and variable-length arguments (*args and **kwargs). Understanding these argument types is important when writing flexible and reusable functions.

Recursion

Recursion is a powerful programming technique in which a function calls itself to solve a problem. A recursive function is a function that calls itself with a modified set of parameters, and the process continues until a base condition is met. Recursion can be used to solve problems that can be broken down into smaller subproblems, and is often used in algorithms for searching, sorting, and tree traversal.

Rapt ‘N Rel

Python Documentation

In Python, a recursive function has two parts: the base case and the recursive case. The base case is the condition under which the function stops calling itself, and returns a value. The recursive case is the condition under which the function calls itself with modified parameters.

Here's an example of a recursive function that calculates the factorial of a number:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

In this example, the base case is when the input parameter n is 0. When n is 0, the function returns 1. The recursive case is when n is greater than 0. In this case, the function calls itself with n-1 as the input parameter, and multiplies the result by n.

To understand how this function works, let's look at the call stack for calculating factorial(3):

```
factorial(3)
3 * factorial(2)
3 * (2 * factorial(1))
3 * (2 * (1 * factorial(0)))
3 * (2 * (1 * 1))
6
```

In this example, the function is called with an input parameter of 3. The function checks if n is 0, which is not the case. The function then calls itself with n-1 as the input parameter, which is 2. The function checks if n is 0, which is not the case, and calls itself again with n-1 as the input parameter, which is 1. The function checks if n is 0, which is not the case, and calls itself again with n-1 as the input parameter, which is 0. This time, the base case is met, and the function returns 1. The function then returns the result of 1 * 1, which is 1, to the previous call. The next call multiplies 2 * 1, which is 2, and returns it to the previous call. Finally, the initial call multiplies 3 * 2, which is 6, and returns the result.

Recursion can be a powerful tool for solving complex problems, but it can also lead to stack overflow errors if not used properly. It's important to make sure that the base case is eventually met, and that the recursive calls are not infinitely nested.

Anonymous Function

In Python, an anonymous function is a function that is defined without a name. Anonymous functions are also called lambda functions, because they are defined using the lambda keyword.

Here's the basic syntax for defining a lambda function:

```
lambda arguments: expression
```

In this syntax, arguments is a comma-separated list of function arguments, and expression is a single expression that is evaluated and returned by the function.

Here's an example of a lambda function that squares its input:

```
square = lambda x: x**2
```

In this example, the lambda function takes one argument x, and returns the square of x.

Lambda functions can be useful when you need to define a small, one-time-use function. They are often used in combination with other functions, such as map(), filter(), and reduce(), to create more complex functionality.

For example, here's how you could use a lambda function with the map() function to create a list of the squares of a list of numbers:

```
numbers = [1, 2, 3, 4, 5]
squares = map(lambda x: x**2, numbers)
print(list(squares))
```

In this example, the map() function applies the lambda function to each element of the numbers list, and returns an iterator that generates the squares of each number. The list() function is then used to convert the iterator to a list that can be printed.

Lambda functions can also be used as key functions in sorting functions such as sorted() and sort(). For example, here's how you could sort a list of tuples based on the second element of each tuple:

```
data = [(3, "apple"), (1, "banana"), (2, "cherry")]
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)
```

In this example, the sorted() function sorts the data list based on the second element of each tuple, which is accessed using the lambda function lambda x: x[1]. The print() function is then used to print the sorted list.

Lambda functions are a convenient way to define small, simple functions without having to give them a name. They are often used in functional programming, where functions are treated as first-class objects that can be passed as arguments to other functions.

In Python, variables can have different scopes depending on where they are defined and used. The three main scopes are global, local, and non-local.

Scope

In Python, the scope refers to the region of the code where a variable or a function is visible and can be accessed. There are three types of scope in Python:

Global Variables

A variable declared outside of a function is called a global variable. Global variables are accessible from anywhere in the code, including inside functions. Global variables can be accessed and modified by any function in the program.

Example:

```
x = 10    # Global variable


def func():
```

```
    print(x)    # Accessing global variable inside
function

func()    # Output: 10
```

Global Keyword

In Python, the global keyword is used to declare a variable inside a function as a global variable. A global variable is a variable that can be accessed from anywhere in the code, including inside functions. Without the global keyword, a variable declared inside a function is considered local to that function and cannot be accessed from outside of the function.

Here's an example of how the global keyword works:

```
x = 10    # Global variable

def func():
    global x    # Declare x as global inside the
function
    x = 5    # Modify global variable inside the
function
    print(x)

func()    # Output: 5
print(x)    # Output: 5
```

In the above example, the global keyword is used to declare x as a global variable inside the function. This means that the x variable inside the function is the same as the x variable declared outside the function. When the value of x is changed inside the function, it also changes the value of the global x variable.

It's important to note that the global keyword should be used with caution, as it can lead to unexpected behavior and make the code harder to debug. It's generally better to avoid using

global variables as much as possible and instead use function arguments and return values to pass data between functions.

Local Variables

A variable declared inside a function is called a local variable. Local variables are only accessible within the function where they are declared. These variables have a local scope and are destroyed once the function exits.
Example:

```
def func():
    x = 5     # Local variable
    print(x)    # Accessing local variable

func()    # Output: 5
```

Non-local Variables

A non-local variable is a variable that is neither global nor local. A non-local variable is declared in a nested function and is accessible to the nested function and any other functions defined within the same scope. Non-local variables can be accessed but not modified by inner functions.

Example:

```
def outer():
    x = 10    # Outer function variable

    def inner():
        nonlocal x    # Declaring non-local variable
        x = 20     # Modifying non-local variable
        print(x)    # Accessing non-local variable
```

```
    inner()
    print(x)    # Accessing modified non-local
variable

outer()    # Output: 20 20
```

In summary, global variables have a global scope and are accessible from anywhere in the code, local variables have a local scope and are only accessible within the function where they are declared, and non-local variables are accessible to nested functions and can be accessed but not modified by inner functions.

Modules

In Python, a module is a file containing Python code that can be imported and used in other Python code. Modules are used to organize code into logical and reusable units, making it easier to maintain and reuse code across projects. A module can contain functions, classes, variables, and other Python code.

To use a module in Python, you need to import it using the import keyword. Here's an example:

```
import math

print(math.sqrt(25))    # Output: 5.0
```

In the above example, the math module is imported using the import keyword, and the sqrt function from the math module is used to calculate the square root of 25.

You can also import specific functions or variables from a module using the from keyword. Here's an example:

```
from math import sqrt

print(sqrt(25))    # Output: 5.0
```

In the above example, the sqrt function is imported from the math module using the from keyword. This allows you to use the sqrt function directly without having to prefix it with the module name.

Python also includes many built-in modules that provide additional functionality, such as the random module for generating random numbers and the datetime module for working with dates and times.

Creating your own modules is also possible in Python. To create a module, you simply need to create a file with a .py extension containing the Python code you want to include in the module. You can then import the module in your Python code using the import keyword.

In summary, a Python module is a file containing Python code that can be imported and used in other Python code. Modules are used to organize code into logical and reusable units, making it easier to maintain and reuse code across projects. Python includes many built-in modules, and it's also possible to create your own modules.

Package

In Python, a package is a collection of modules grouped together in a directory hierarchy. Packages allow for the organization and management of large Python projects by providing a way to divide the code into logical units.

A package is simply a directory that contains an __init__.py file, which indicates to Python that the directory should be treated as a package. The __init__.py file can contain Python code that is executed when the package is imported, such as defining variables or importing modules.

Here's an example of a simple package structure:

```
mypackage/
    __init__.py
    module1.py
    module2.py
```

In the above example, the mypackage directory is a package containing two modules, module1.py and module2.py. The __init__.py file is empty in this example, but it could contain any valid Python code.

To use a module from a package, you need to import it using dot notation. For example, to import the module1 module from the mypackage package, you would use the following code:

```
from mypackage import module1


module1.my_function()
```

In the above example, the my_function function from the module1 module is called using dot notation.

You can also import multiple modules from a package using the from keyword. Here's an example:

```
from mypackage import module1, module2


module1.my_function()
module2.my_other_function()
```

In the above example, both the module1 and module2 modules are imported from the mypackage package using the from keyword.

In addition to organizing code, packages can also be used to distribute and install Python code using tools such as pip and setuptools.

In summary, a Python package is a collection of modules grouped together in a directory hierarchy. Packages allow for the organization and management of large Python projects by providing a way to divide the code into logical units. Packages are created by adding an __init__.py file to a directory, and modules from a package are imported using dot notation.

Rapt ‘N Rel

Python Documentation

File Handling In Python

File handling in Python allows you to manipulate files on your computer's file system.

Operations

File operations in Python refer to various actions that you can perform on a file, such as creating, opening, reading, writing, and deleting files.

Creating a file

To create a new file in Python, you can use the open() function with the "w" mode (write mode). Here's an example:

```
file = open("new_file.txt", "w")
```

In the above example, the open() function is used to create a new file called new_file.txt in write mode ("w"), and a file object is returned.

Opening a file

To open an existing file in Python, you can use the open() function with the "r" mode (read mode). Here's an example:

```
file = open("existing_file.txt", "r")
```

In the above example, the open() function is used to open an existing file called existing_file.txt in read mode ("r"), and a file object is returned.

Reading a file

To read the contents of a file in Python, you can use the read() method on the file object. Here's an example:

```
file = open("example.txt", "r")
```

```
content = file.read()
print(content)
file.close()
```

In the above example, the open() function is used to open an existing file called example.txt in read mode ("r"), and the read() method is used to read the contents of the file into the content variable. Finally, the close() method is called to close the file.

Writing to a file

To write to a file in Python, you can use the write() method on the file object. Here's an example:

```
file = open("new_file.txt", "w")
file.write("This is some text that will be written to the file.")
file.close()
```

In the above example, the open() function is used to create a new file called new_file.txt in write mode ("w"), and the write() method is used to write some text to the file. Finally, the close() method is called to close the file.

Appending to a file

To append to an existing file in Python, you can use the open() function with the "a" mode (append mode) and the write() method. Here's an example:

```
file = open("existing_file.txt", "a")
file.write("This text will be appended to the end of the file.")
file.close()
```

In the above example, the open() function is used to open an existing file called existing_file.txt in append mode ("a"), and the write() method is used to append some text to the end of the file. Finally, the close() method is called to close the file.

Deleting a file

To delete a file in Python, you can use the os module and the remove() function. Here's an example:

```
import os


os.remove("file_to_delete.txt")
```

In the above example, the os.remove() function is used to delete a file called file_to_delete.txt.

It's important to note that when working with files, you should always close the file after you're done with it using the close() method. This ensures that any changes you made to the file are saved and that the

Directory

In Python, a file directory is a collection of files and subdirectories that are organized in a hierarchical structure. The top-level directory is known as the root directory, and all other directories are located within this root directory.

Python provides several built-in modules that you can use to work with file directories. Here are some of the most commonly used modules:

os module

The os module provides a set of functions that allow you to work with file directories in a platform-independent way. Some of the most commonly used functions include:

- os.getcwd(): returns the current working directory
- os.chdir(path): changes the current working directory to the specified path

- os.mkdir(path): creates a new directory with the specified path
- os.makedirs(path): creates a new directory and all parent directories with the specified path
- os.listdir(path): returns a list of files and directories in the specified path
- os.rmdir(path): removes the directory with the specified path (directory must be empty)
- os.removedirs(path): removes the directory with the specified path and all parent directories (all directories must be empty)

Here's an example of how you might use the os module to create a new directory:

```
import os

# create a new directory
os.mkdir("my_directory")

# change the current working directory to the new
directory
os.chdir("my_directory")

# create a new file in the new directory
with open("my_file.txt", "w") as file:
    file.write("This is some text.")

# print the contents of the file
with open("my_file.txt", "r") as file:
    print(file.read())
```

Rapt ‘N Rel

Python Documentation

pathlib module

The pathlib module provides an object-oriented interface for working with file directories. This module was introduced in Python 3.4 and is preferred over the os module for many tasks.

Here's an example of how you might use the pathlib module to create a new directory:

```
import pathlib

# create a new directory
pathlib.Path("my_directory").mkdir()

# change the current working directory to the new directory
os.chdir("my_directory")

# create a new file in the new directory
with open("my_file.txt", "w") as file:
    file.write("This is some text.")

# print the contents of the file
with open("my_file.txt", "r") as file:
    print(file.read())
```

shutil module

The shutil module provides a set of high-level operations for working with file directories, such as copying and moving files and directories. Here's an example of how you might use the shutil module to copy a directory:

```
import shutil
```

```
# create a new directory
os.mkdir("my_directory")

# create a new file in the new directory
with open("my_directory/my_file.txt", "w") as file:
    file.write("This is some text.")

# copy the directory to a new location
shutil.copytree("my_directory", "my_copy")

# print the contents of the copied file
with open("my_copy/my_file.txt", "r") as file:
    print(file.read())
```

Overall, working with file directories in Python is made easy by the built-in modules and functions provided by the language. Whether you need to create, copy, move, or delete files and directories, Python has you covered.

Exception

In Python, an exception is an error that occurs during the execution of a program. When an exception is raised, the program stops running and Python's default behavior is to print an error message that describes the exception and its location in the code. However, you can also handle exceptions in your code using the try-except statement.

Here's an example of how you might use the try-except statement to handle an exception:

```
try:
    x = 1 / 0
```

```
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

In this example, the code inside the try block attempts to divide 1 by 0, which will raise a ZeroDivisionError exception. However, because the code is wrapped in a try block, the program won't stop running. Instead, the except block is executed and the message "Cannot divide by zero!" is printed.

Python provides several built-in exception types that you can use to handle specific types of errors, such as:

- ZeroDivisionError: raised when attempting to divide by zero
- TypeError: raised when an operation or function is applied to an object of inappropriate type
- NameError: raised when a variable or name is not found in the current scope
- ValueError: raised when a built-in operation or function receives an argument of the correct type but an inappropriate value

You can also create your own custom exception types by subclassing the built-in Exception class.

Here's an example of how you might define and raise a custom exception:

```
class MyCustomException(Exception):
    pass

def my_function(x):
    if x < 0:
        raise MyCustomException("Input must be non-
negative!")
    return x ** 2

try:
```

```
    result = my_function(-1)
except MyCustomException as e:
    print(e)
```

In this example, the MyCustomException class is defined by subclassing the built-in Exception class. The my_function function raises an instance of this custom exception if the input is negative. Finally, the code inside the try block calls the function with a negative input, causing the exception to be raised and the message "Input must be non-negative!" to be printed.

Overall, exceptions are an important part of Python programming because they allow you to gracefully handle errors in your code and provide informative error messages to users. By using the try-except statement and the built-in and custom exception types provided by Python, you can write code that is more robust and less prone to crashing.

Exception Handling

In Python, exception handling allows you to gracefully handle errors that occur during the execution of your program. Instead of simply crashing when an error occurs, you can catch the error and take appropriate action, such as displaying an error message to the user or retrying the operation that caused the error.

In Python, exception handling is done using the try-except statement. Here's the basic syntax:

```
try:
    # code that might raise an exception
except ExceptionType:
    # code to handle the exception
```

In this example, the code inside the try block is executed. If an exception of type ExceptionType is raised, the code inside the except block is executed instead. The except block can contain any code that you want to run in response to the exception, such as displaying an error message or retrying the operation that caused the error.

Here's an example that demonstrates how you might use exception handling to catch a ZeroDivisionError:

```
try:
```

```
    x = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

In this example, the code inside the try block attempts to divide 1 by 0, which will raise a ZeroDivisionError. However, because the code is wrapped in a try block, the program won't crash. Instead, the code inside the except block is executed, and the message "Cannot divide by zero!" is printed.

Python provides several built-in exception types that you can use to handle specific types of errors, such as:

- ZeroDivisionError: raised when attempting to divide by zero
- TypeError: raised when an operation or function is applied to an object of inappropriate type
- NameError: raised when a variable or name is not found in the current scope
- ValueError: raised when a built-in operation or function receives an argument of the correct type but an inappropriate value

You can also create your own custom exception types by subclassing the built-in Exception class.

Finally, it's important to note that you can use multiple except blocks to handle different types of exceptions. For example:

```
try:
    # code that might raise an exception
except ExceptionType1:
    # code to handle ExceptionType1
except ExceptionType2:
    # code to handle ExceptionType2
except:
    # code to handle all other exceptions
```

In this example, the first except block handles exceptions of type ExceptionType1, the second except block handles exceptions of type ExceptionType2, and the third except block handles all other exceptions that are not caught by the first two except blocks. This can be useful if you want to handle different types of errors in different ways, or if you want to provide more detailed error messages for specific types of errors.

User-defined Exceptions

In Python, you can define your own custom exception classes by subclassing the built-in Exception class. This allows you to create exceptions that are tailored to the specific needs of your program and provide more detailed information about the error that occurred.

To define a custom exception class, you simply create a new class that inherits from the Exception class, and then add any additional functionality that you need. Here's an example:

```
class MyException(Exception):
    pass


raise MyException("Something went wrong.")
```

In this example, the MyException class is defined by subclassing the built-in Exception class. The pass keyword is used to indicate that the class has no additional functionality beyond what is inherited from Exception. Finally, the raise statement is used to raise an instance of the MyException class, along with a message that describes the error that occurred.

You can also add additional functionality to your custom exception class, such as attributes or methods that provide more information about the error. For example:

```
class MyException(Exception):
    def __init__(self, message, code):
        self.message = message
        self.code = code

    def __str__(self):
```

```
        return f"{self.message} (error code:
{self.code})"


raise MyException("Something went wrong.", 1234)
```

In this example, the MyException class is defined with two additional attributes: message and code. These attributes are set in the init method, which is called when an instance of the class is created. The str method is also defined to provide a string representation of the exception, which includes both the message and the error code. Finally, an instance of the MyException class is raised with a message and error code.

By defining your own custom exception classes, you can create more informative and user-friendly error messages that provide greater context about the errors that occur in your program. This can be particularly useful in larger, more complex programs where it may be difficult to determine the cause of an error without additional information.

Object-Oriented Programming In Python

Object-oriented programming (OOP) is a programming paradigm that emphasizes the use of objects, which are instances of classes, to represent real-world entities and concepts. In Python, OOP is supported through classes, which are defined using the class keyword.

Classes And Objects

A class in Python provides a blueprint for creating objects that have both attributes and behaviors. For instance, if we consider a parrot as an object, it has attributes such as name, age, color, etc. and behaviors such as dancing, singing, etc. A class serves as a blueprint for creating such an object.

Rapt ‘N Rel

Python Documentation

Classes

In Python, classes are used to create objects that encapsulate data and behavior. A class is a blueprint for creating objects, which defines a set of attributes and methods that are shared by all instances of the class.

To define a class in Python, you use the class keyword followed by the name of the class, like this:

```
class MyClass:
    pass
```

This defines a simple class called MyClass that doesn't have any attributes or methods yet. To create an instance of a class, you call the class as if it were a function, like this:

```
my_object = MyClass()
```

This creates a new object of the MyClass type and assigns it to the variable my_object. You can then access the object's attributes and methods using the dot notation:

```
my_object.my_attribute = "Hello, world!"
print(my_object.my_attribute)
```

This would output Hello, world!.

To define attributes and methods for a class, you use the same syntax as you would for defining variables and functions, but you indent the code inside the class definition. Here's an example of a class with attributes and methods:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
```

```
        print(f"Hello, my name is {self.name} and I
am {self.age} years old.")

person = Person("Alice", 25)
person.greet()  # output: Hello, my name is Alice and
I am 25 years old.
```

In this example, we define a Person class with two attributes, name and age, and one method, greet(). The __init__() method is a special method called the constructor, which is used to initialize the object's state when it is created. The self parameter refers to the object being created, and is used to access its attributes and methods.

Python also supports inheritance, which allows you to create new classes that are based on existing classes. To create a subclass, you define a new class that inherits from the parent class, like this:

```
class Student(Person):
    def __init__(self, name, age, major):
        super().__init__(name, age)
        self.major = major

    def study(self):
        print(f"{self.name} is studying
{self.major}.")

student = Student("Bob", 20, "Computer Science")
student.greet()  # output: Hello, my name is Bob and
I am 20 years old.
student.study()  # output: Bob is studying Computer
Science.
```

In this example, we define a Student class that inherits from the Person class. The Student class has an additional attribute major and method study(), which are specific to students. We use the super() function to call the parent class's constructor and initialize the name and age attributes.

Overall, classes are a powerful feature of Python that allow you to write code that is modular, reusable, and easy to understand. By using classes, you can represent complex systems and structures in a clear and concise manner, and take advantage of Python's powerful built-in features and libraries to solve a wide range of problems.

Objects

In Python, an object is an instance of a class, which contains data (attributes) and functions (methods) that can operate on that data. Everything in Python is an object, including built-in data types like integers, strings, and lists.

Objects in Python are created using classes, which are essentially templates or blueprints that define the attributes and methods that the objects will have. When a class is instantiated, an object is created with the same attributes and methods defined in the class.

For example, if we have a class called Person with attributes name, age, and gender, and methods speak and walk, we can create an object person1 of type Person with specific values for the attributes:

```
class Person:
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    def speak(self):
        print("Hello, my name is", self.name)

    def walk(self):
        print(self.name, "is walking")
```

```
person1 = Person("Alice", 25, "female")
```

In this example, we create a Person object named person1 with the name "Alice", age 25, and gender "female". We can access the attributes and methods of this object using dot notation:

```
print(person1.name)   # output: Alice
person1.speak()       # output: Hello, my name is Alice
person1.walk()        # output: Alice is walking
```

In addition to the attributes and methods defined in the class, objects in Python also have some built-in methods that can be used to manipulate them. For example, the dir() function can be used to list all the attributes and methods of an object:

```
print(dir(person1))
```

This will output a list of all the attributes and methods of the person1 object, including those inherited from the object class.

Inheritance

In Python, inheritance is a mechanism that allows a new class (called the derived class or subclass) to be based on an existing class (called the base class or superclass), inheriting all of its attributes and methods. The derived class can then add its own attributes and methods, and can also override or modify those inherited from the base class.

Inheritance is a powerful feature of object-oriented programming that enables code reuse and promotes modularity. It allows us to create classes that are specialized versions of existing classes, with specific behaviors and characteristics.

To create a subclass in Python, we use the syntax class DerivedClass(BaseClass):. This tells Python that the new class should inherit from the specified base class. For example:

```
class Animal:
```

```
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("I am an animal.")

class Cat(Animal):
    def speak(self):
        print("Meow.")

cat1 = Cat("Fluffy")
print(cat1.name)    # output: Fluffy
cat1.speak()        # output: Meow.
```

In this example, we define a base class Animal with an __init__ method that initializes the name attribute, and a speak method that prints a generic message. We then define a subclass Cat that inherits from Animal, and overrides the speak method with a more specific message.

When we create an instance of the Cat class, it has all the attributes and methods of the Animal class, plus any additional attributes or methods defined in the Cat class. In this case, the Cat class has a name attribute inherited from Animal, and a speak method that overrides the one inherited from Animal.

Inheritance can be used to create complex class hierarchies, with multiple levels of inheritance and multiple derived classes. It allows us to create classes that are more specific and specialized than their base classes, and to avoid code duplication by reusing existing code.

Multiple Inheritance

Multiple inheritance is a feature of object-oriented programming that allows a subclass to inherit from more than one base class. In Python, we can implement multiple inheritance by specifying multiple base classes in the class definition.

The syntax for creating a class with multiple inheritance in Python is as follows:

```
class DerivedClass(BaseClass1, BaseClass2, ...):
    ...
```

In this syntax, DerivedClass is the name of the new class we are defining, and BaseClass1, BaseClass2, and so on are the names of the base classes that the new class will inherit from.

When a subclass inherits from multiple base classes, it inherits all of their attributes and methods. If two or more base classes have methods with the same name, the method of the first base class listed in the class definition is used. This is called method resolution order (MRO).

Here's an example of multiple inheritance in Python:

```
class Animal:
    def speak(self):
        print("I am an animal.")

class Mammal:
    def feed_milk(self):
        print("I feed milk.")

class Cat(Animal, Mammal):
    def speak(self):
        print("Meow.")

cat1 = Cat()
cat1.speak()        # output: Meow.
cat1.feed_milk()    # output: I feed milk.
```

In this example, we define two base classes Animal and Mammal, and a subclass Cat that inherits from both Animal and Mammal. The Cat class overrides the speak method inherited from Animal, and inherits the feed_milk method from Mammal.

Multiple inheritance can be a powerful tool for creating complex class hierarchies and building modular, reusable code. However, it can also lead to complex code and potential conflicts between base classes. Therefore, it should be used judiciously and with care.

Operator Overloading

Operator overloading is a feature of object-oriented programming in Python that allows you to define how built-in operators behave when applied to user-defined objects. This means that you can customize the behavior of operators like +, -, *, / and many others for your own classes and objects.

In Python, operator overloading is implemented using special methods or "magic methods". These methods have names starting and ending with double underscores, such as __add__, __sub__, __mul__, __div__, etc. When you use one of these operators on an object, Python looks for the appropriate magic method and uses it to perform the operation.

Here is an example of how operator overloading works in Python:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y +
other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y -
other.y)
```

```
    def __mul__(self, other):
        return Vector(self.x * other, self.y * other)

    def __rmul__(self, other):
        return Vector(self.x * other, self.y * other)

    def __str__(self):
        return f'({self.x}, {self.y})'

v1 = Vector(1, 2)
v2 = Vector(3, 4)

print(v1 + v2)  # Output: (4, 6)
print(v2 - v1)  # Output: (2, 2)
print(v1 * 2)   # Output: (2, 4)
print(2 * v1)   # Output: (2, 4)
```

In this example, we have defined a Vector class that has magic methods to overload the +, -, * and *= operators. When we use these operators on instances of the Vector class, Python automatically calls the appropriate magic method to perform the operation.

Operator overloading can be a powerful tool for creating more expressive and readable code, but it should be used judiciously to avoid making code too complex or difficult to understand.

Rapt ‘N Rel

Python Documentation

Advanced Concepts In Python

Pythonic Features

Iterator

In Python, an iterator is an object that allows you to iterate through a collection of data, such as a list, tuple or dictionary. It provides a way to access elements of a collection one by one, without needing to know the underlying implementation of the collection.

An iterator is defined by implementing two special methods: __iter__() and __next__(). The __iter__() method returns the iterator object itself, and the __next__() method returns the next value in the collection. When there are no more elements in the collection, the __next__() method should raise the StopIteration exception.

Here's an example of how to create an iterator in Python:

```
class MyIterator:
    def __init__(self, data):
        self.index = 0
        self.data = data

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.data):
            raise StopIteration
        value = self.data[self.index]
        self.index += 1
        return value
```

```
my_list = [1, 2, 3, 4, 5]
my_iterator = MyIterator(my_list)

for item in my_iterator:
    print(item)
```

In this example, we define a MyIterator class that implements the __iter__() and __next__() methods. We create an instance of this class with a list of integers, and then use a for loop to iterate through the items in the list using the iterator.

Note that in Python, many built-in data types are iterable, including lists, tuples, dictionaries, and strings. You can also use the iter() function to create an iterator from an iterable object, and the next() function to get the next value from the iterator.

Iterators are a powerful and flexible tool in Python, and they can be used to create custom data types and implement complex algorithms. By using iterators, you can write more efficient and expressive code that is easier to read and maintain.

Generator

In Python, a generator is a special type of iterator that allows you to generate a sequence of values on the fly, rather than storing them all in memory at once. This can be useful for working with large datasets or for generating an infinite sequence of values.

Generators are defined using a special kind of function that uses the yield keyword instead of return to generate a series of values. Each time the function is called, it generates the next value in the sequence and then pauses until the next value is requested.

Here's an example of how to create a generator in Python:

```
def my_generator():
    for i in range(10):
        yield i
```

```
my_sequence = my_generator()

for item in my_sequence:
    print(item)
```

In this example, we define a my_generator() function that uses the yield keyword to generate a sequence of numbers from 0 to 9. We then create a generator object from this function and use a for loop to iterate through the sequence of values.

One of the benefits of using generators in Python is that they can be used to generate infinite sequences of values. For example, here's a generator that generates an infinite sequence of even numbers:

```
def even_numbers():
    i = 0
    while True:
        yield i
        i += 2

my_sequence = even_numbers()

for item in my_sequence:
    print(item)
```

In this example, we define an even_numbers() function that uses an infinite loop to generate a sequence of even numbers. We then create a generator object from this function and use a for loop to iterate through the sequence of values.

Generators are a powerful and flexible tool in Python, and they can be used to create custom data types and implement complex algorithms. By using generators, you can write more efficient and expressive code that is easier to read and maintain.

Rapt ‘N Rel

Python Documentation

Decorator

In Python, a decorator is a special type of function that can modify or enhance the behavior of other functions or classes. Decorators are a key feature of Python's functional programming capabilities, and they can be used to add functionality to existing code without modifying the original source.

A decorator is defined using the @ symbol followed by the name of the decorator function. The decorator function takes another function as its argument and returns a modified version of that function. Here's an example of a simple decorator that adds logging functionality to a function:

```
def log_decorator(func):
    def wrapper(*args, **kwargs):
        print("Calling function", func.__name__)
        result = func(*args, **kwargs)
        print("Function", func.__name__, "returned",
result)
        return result
    return wrapper

@log_decorator
def add_numbers(a, b):
    return a + b

result = add_numbers(2, 3)
print(result)
```

In this example, we define a log_decorator() function that takes another function as its argument and returns a modified version of that function. The modified function prints a log message before and after calling the original function.

We then define a add_numbers() function and apply the @log_decorator decorator to it. This modifies the behavior of the add_numbers() function by adding logging functionality.

Finally, we call the add_numbers() function with two arguments and print the result.

Decorators can be used for a wide range of purposes in Python, including:

- Adding authentication or authorization checks to a function or method
- Implementing caching or memoization for expensive functions
- Adding validation or error handling to a function or method
- Measuring the performance of a function or method
- Implementing retry or fallback behavior for a function or method

By using decorators, you can add powerful and flexible functionality to your code without cluttering your source with boilerplate code. Decorators are a key feature of Python's functional programming capabilities, and they are a valuable tool for building robust and extensible software systems.

Pythonic Features II

Closure

In Python, a closure is a function that retains access to variables in its enclosing lexical scope, even when the function is called outside that scope. This allows the function to "remember" the values of those variables and use them when it is called later.

Here's an example of a closure in Python:

```
def make_adder(x):
    def adder(y):
        return x + y
    return adder

add5 = make_adder(5)
add10 = make_adder(10)
```

```
print(add5(3))   # Output: 8
print(add10(3))   # Output: 13
```

In this example, we define a function called make_adder() that takes a number x and returns a function called adder() that takes another number y and returns the sum of x and y. The adder() function is a closure because it retains access to the x variable from its enclosing scope.

We then use the make_adder() function to create two new functions: add5 and add10. These functions are closures because they remember the values of x that were passed to make_adder() when they were created.

Finally, we call the add5(3) and add10(3) functions, which return the sums 8 and 13, respectively.

Closures are a powerful feature of Python that allow you to write more flexible and expressive code. They can be used to implement many common programming patterns, such as factory functions, currying, and memoization.

Property

In Python, a property is a special kind of attribute that allows you to define custom getter, setter, and deleter methods for an object's attribute. It provides a way to control access to the attribute and perform additional actions when it is accessed or modified.

Here's an example of using a property in Python:

```
class Rectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height

    @property
    def width(self):
        return self._width
```

```
    @width.setter
    def width(self, value):
        if value <= 0:
            raise ValueError("Width must be
positive")
        self._width = value

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, value):
        if value <= 0:
            raise ValueError("Height must be
positive")
        self._height = value

    def area(self):
        return self._width * self._height
```

In this example, we define a class called Rectangle with two attributes, _width and _height, and a method called area() that calculates the area of the rectangle.

We also define two properties, width and height, using the @property decorator. These properties allow us to get the values of the _width and _height attributes using the rectangle.width and rectangle.height syntax, respectively.

We also define setters for the width and height properties using the @width.setter and @height.setter decorators. These setters allow us to modify the values of the _width and

_height attributes using the rectangle.width = value and rectangle.height = value syntax, respectively. In these setters, we also perform validation to ensure that the values are positive.

With this implementation, we can create a Rectangle object and manipulate its properties to calculate its area:

```
rectangle = Rectangle(3, 4)
print(rectangle.area())  # Output: 12

rectangle.width = 5
rectangle.height = 6
print(rectangle.area())  # Output: 30

rectangle.width = -1  # Raises ValueError
```

The property feature in Python allows us to create more expressive and flexible classes, making it easier to work with objects in a more intuitive and natural way.

RegEx

Regular expressions, commonly known as regex or regexp, is a sequence of characters that define a search pattern. It is a powerful tool used for matching and manipulating text. Python provides support for regular expressions through the re module.

The re module in Python allows you to use regular expressions in your code. It provides several functions and methods to work with regular expressions, such as match(), search(), findall(), sub(), split(), and more.

To use regular expressions in Python, you first need to import the re module. Once imported, you can use the functions and methods provided by the module.

Regular expressions in Python use a syntax that is similar to other programming languages. Here are some commonly used symbols and expressions:

- . : matches any single character except for a newline character.
- ^ : matches the beginning of a string.

- $: matches the end of a string.
- * : matches zero or more occurrences of the preceding character.
- + : matches one or more occurrences of the preceding character.
- ? : matches zero or one occurrence of the preceding character.
- [ ] : matches any one of the characters enclosed in the square brackets.
- [^] : matches any one character that is not enclosed in the square brackets.
- () : creates a group that can be referenced later.

Here's a brief overview of some commonly used functions and methods in the re module:

re.match(pattern, string):

searches for the pattern at the beginning of the string and returns a match object if it finds a match. Otherwise, it returns None.

Here's an example of using re.match() in Python:

```
import re

text = "Hello, world!"

pattern = r"^Hello"  # matches "Hello" at the
beginning of the string

match = re.match(pattern, text)

if match:
    print("Match found:", match.group())
else:
```

```
print("No match found")
```

In this example, we start by importing the re module which provides support for regular expressions.

We define a text variable that contains the string we want to search for a match in, and a pattern variable that contains the regular expression pattern we want to match.

The r in front of the pattern string indicates that it should be treated as a "raw string", which means that backslashes won't be interpreted as escape sequences.

We then call re.match(pattern, text) to attempt to match the pattern at the beginning of the string. The ^ character in the pattern matches the start of the string.

If a match is found, re.match() returns a Match object that contains information about the match. We can access the matched text using the group() method of the Match object.

In this example, we print the matched text if a match is found, or a message indicating that no match was found if the pattern doesn't match the text.

re.search(pattern, string):

searches for the pattern anywhere in the string and returns a match object if it finds a match. Otherwise, it returns None.

Here's an example of using re.search():

```
import re

# search for a pattern in a string
string = "The quick brown fox jumps over the lazy dog."
pattern = "fox"

match = re.search(pattern, string)
```

```
if match:
    print("Found the pattern:", match.group())
else:
    print("The pattern was not found.")

# Output:
# Found the pattern: fox
```

In this example, we're searching for the string "fox" in the string "The quick brown fox jumps over the lazy dog.". We use the re.search() function to search for the pattern and assign the result to the variable match.

If the pattern is found, re.search() returns a match object. We can then call the group() method on the match object to return the matching string. If the pattern is not found, re.search() returns None.

re.findall(pattern, string):

searches for all occurrences of the pattern in the string and returns a list of all matches.

here's an example of using re.findall():

```
import re

# find all occurrences of a pattern in a string
string = "The quick brown fox jumps over the lazy dog."
pattern = "\w+"

matches = re.findall(pattern, string)

print(matches)
```

```
# Output: ['The', 'quick', 'brown', 'fox', 'jumps',
'over', 'the', 'lazy', 'dog']
```

In this example, we're searching for all occurrences of one or more word characters (i.e., letters, digits, or underscores) in the string "The quick brown fox jumps over the lazy dog.". We use the re.findall() function to find all matches of the pattern and assign the result to the variable matches.

re.findall() returns a list of all matches found in the string. Each item in the list is a string representing a match of the pattern in the input string.

re.sub(pattern, repl, string):

replaces all occurrences of the pattern in the string with the replacement string specified by repl.

Here's an example of using re.sub():

```
import re

# replace all occurrences of a pattern in a string
string = "The quick brown fox jumps over the lazy
dog."
pattern = "\s+"

new_string = re.sub(pattern, "_", string)

print(new_string)

# Output:
The_quick_brown_fox_jumps_over_the_lazy_dog.
```

In this example, we're replacing all occurrences of one or more whitespace characters with an underscore in the string "The quick brown fox jumps over the lazy dog.". We use the re.sub() function to perform the substitution and assign the result to the variable new_string.

re.sub() returns a new string with all occurrences of the pattern replaced by the specified replacement string.

Each occurrence of the pattern in the original string is replaced by the specified replacement string (in this case, an underscore). Note that the original string is not modified by the re.sub() function.

re.split(pattern, string):

splits the string into a list of substrings at the occurrences of the pattern.

Here's an example of using re.split():

```
import re

# split a string by a pattern
string = "The quick brown fox jumps over the lazy dog."
pattern = "\s+"

matches = re.split(pattern, string)

print(matches)

# Output: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog.']
```

In this example, we're splitting the string "The quick brown fox jumps over the lazy dog." by one or more whitespace characters. We use the re.split() function to split the string into a list of substrings and assign the result to the variable matches.

re.split() returns a list of substrings split by the given pattern.

Each item in the list is a substring split by the pattern in the input string. Note that the final substring includes the period at the end of the sentence because it was not matched by the pattern.

Regular expressions can be complex and powerful, but they can also be difficult to read and write. It's important to carefully test and debug your regular expressions to ensure they are working as intended.

Date & Time

In Python, the datetime module provides classes to work with dates and times. The date class represents a date (year, month, day) and the time class represents a time (hour, minute, second, microsecond). The datetime class is a combination of date and time.

You can create a date or time object using the respective constructor date(year, month, day) or time(hour=0, minute=0, second=0, microsecond=0). To create a datetime object, you can use the constructor datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0).

The datetime module also provides functions to format a date or time as a string using the strftime method, and to parse a string as a date or time object using the strptime method.

To work with time zones, you can use the pytz library which provides timezone definitions and conversion functions. The datetime module provides a timezone class which can be used to create a timezone object.

datetime Module

The datetime module in Python is a built-in module that provides classes for working with dates and times. It includes three main classes:

- date class: This class represents a date with attributes such as year, month, and day. You can create a date object using the constructor date(year, month, day).

- time class: This class represents a time with attributes such as hour, minute, second, and microsecond. You can create a time object using the constructor time(hour=0, minute=0, second=0, microsecond=0).

- datetime class: This class represents a combination of date and time with attributes such as year, month, day, hour, minute, second, and microsecond. You can create a datetime object using the constructor datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0).

The datetime module also provides various functions and methods for working with dates and times, such as:

- datetime.now(): This function returns the current date and time as a datetime object.

- datetime.date(): This method returns the date part of a datetime object as a date object.

- datetime.time(): This method returns the time part of a datetime object as a time object.

- datetime.strftime(): This method formats a datetime object as a string according to a specified format.

- datetime.strptime(): This method parses a string into a datetime object according to a specified format.

- datetime.timedelta(): This class represents a duration or difference between two dates or times.

The datetime module also includes support for time zones, including the timezone class, which can be used to create a timezone object. The pytz library provides timezone definitions and conversion functions that can be used in conjunction with the datetime module.

Here are some code examples for the datetime module in Python:

```
#1 Creating a datetime object:
import datetime
# Create a datetime object for the current date and time
now = datetime.datetime.now()
```

```
# Create a datetime object for a specific date and
time
dt = datetime.datetime(2022, 3, 7, 14, 30, 0)
print(now)
print(dt)
# Output:
# 2023-03-03 17:25:23.977295
# 2022-03-07 14:30:00


#2 Formatting datetime objects as strings:
import datetime
# Create a datetime object for the current date and
time
now = datetime.datetime.now()
# Format the datetime object as a string
date_string = now.strftime("%Y-%m-%d %H:%M:%S")
print(date_string)
# Output: 2023-03-03 17:30:12


#3 Parsing datetime strings into datetime objects:
import datetime
# Parse a datetime string into a datetime object
dt_string = "2022-03-07 14:30:00"
dt = datetime.datetime.strptime(dt_string, "%Y-%m-%d
%H:%M:%S")
```

```
print(dt)
# Output: 2022-03-07 14:30:00


#4 Performing arithmetic with datetime objects:
import datetime
# Create a datetime object for the current date and
time
now = datetime.datetime.now()
# Add one day to the datetime object
tomorrow = now + datetime.timedelta(days=1)
# Subtract one week from the datetime object
last_week = now - datetime.timedelta(weeks=1)
print(now)
print(tomorrow)
print(last_week)
# Output:
# 2023-03-03 17:35:29.120166
# 2023-03-04 17:35:29.120166
# 2023-02-24 17:35:29.120166
```

datetime.strftime()

In Python, the strftime() method in the datetime module is used to format a date or time object into a string representation. The method takes a format string as an argument and returns a string that represents the date and time object in the specified format.

The strftime() method uses format codes to represent various components of the date and time object. Here are some common format codes:

- %Y: Four-digit year
- %m: Month (01-12)
- %d: Day of the month (01-31)
- %H: Hour (00-23)
- %M: Minute (00-59)
- %S: Second (00-59)

Here is an example of how to use the strftime() method to format a datetime object:

```
import datetime

# create a datetime object for the current date and time
now = datetime.datetime.now()

# format the datetime object as a string
formatted_date = now.strftime("%Y-%m-%d %H:%M:%S")

# print the formatted date string
print(formatted_date)

# Output: 2022-03-04 09:53:41
```

In this example, we create a datetime object using the now() method, which returns the current date and time. We then use the strftime() method to format the datetime object as a string in the format of YYYY-MM-DD HH:MM:SS. Finally, we print the formatted date string.

This is just one example of how to use the strftime() method to format a datetime object. There are many other format codes and formatting options available that allow you to customize the output string to your specific needs.

datetime.strptime()

In Python's datetime module, strptime() is a method that converts a string to a datetime object. The name strptime stands for "string parse time".

The strptime() method takes two arguments: a string representing a date and time, and a format string specifying the format of the given date and time string. The format string uses directives to specify the format of each component of the datetime string.

Here's the basic syntax for using strptime():

```
datetime.datetime.strptime(date_string, format)
```

Where:

- date_string: The date and time string that needs to be converted to a datetime object.
- format: The format string specifying the format of the date and time string.

Here's an example of how to use strptime() to convert a string to a datetime object:

```
import datetime

date_string = '2022-03-01'
format_string = '%Y-%m-%d'

datetime_obj =
datetime.datetime.strptime(date_string,
format_string)
print(datetime_obj)

# Output: 2022-03-01 00:00:00
```

In the above example, we have a string date_string representing a date and we want to convert it to a datetime object. We pass the string to strptime() method along with a format string format_string which specifies the format of the date string. In this case, %Y-%m-%d specifies that the year is in the four-digit format, the month is represented as a zero-padded two-digit number, and the day is represented as a zero-padded two-digit number.

The strptime() method returns a datetime object representing the given date and time string.

Current Date & Time

The Python datetime module provides several functions and classes to work with date and time values. One of the simplest and most useful functions in the datetime module is the datetime.now() function, which returns the current date and time.

The datetime.now() function takes an optional timezone argument, which can be used to get the current date and time in a specific timezone. If no timezone argument is provided, the function returns the date and time in the local timezone of the computer running the Python script.

Here is an example of using the datetime.now() function to get the current date and time:

```
import datetime

now = datetime.datetime.now()

print("Current date and time: ")
print(now)

# Output:
# Current date and time: 2023-03-05 10:34:29.510758
```

You can also use the strftime() method to format the date and time in a specific way. Here's an example:

```
import datetime
```

```
now = datetime.datetime.now()

formatted_date = now.strftime("%Y-%m-%d %H:%M:%S")

print("Formatted date and time: ")
print(formatted_date)
# Output:
# Formatted date and time: 2023-03-05 10:34:29
```

Note that the %Y, %m, %d, %H, %M, and %S are format codes that represent the year, month, day, hour, minute, and second, respectively.

Current Time

In Python, the datetime module provides various classes and methods to work with dates and times. To get the current time, you can use the datetime.now() method, which returns a datetime object representing the current local date and time.

Here's an example:

```
import datetime

now = datetime.datetime.now()
print(now.time())

# Output: 11:22:33.444555
```

In this example, we first import the datetime module. We then call the datetime.now() method to get the current local date and time, which is stored in the now variable as a datetime object. Finally, we use the time() method to extract the time portion of the datetime object and print it to the console.

Note that the time() method returns a time object, which represents the time of day, without a date or time zone.

Rapt ‘N Rel

Python Documentation

Timestamp To datetime

In Python, the datetime module provides various classes and methods to work with dates and times. If you have a timestamp value and want to convert it to a datetime object, you can use the datetime.fromtimestamp() method.

Here's an example:

```
import datetime

timestamp = 1615190435.0
dt_object =
datetime.datetime.fromtimestamp(timestamp)
print("Datetime object:", dt_object)

# Output:
# Datetime object: 2021-03-08 11:07:15
```

In this example, we first import the datetime module. We then create a variable timestamp with a Unix timestamp value of 1615190435.0, which represents the date and time 2021-03-08 11:07:15 in UTC.

We then use the datetime.fromtimestamp() method to convert the timestamp to a datetime object in the local timezone. The resulting datetime object is stored in the dt_object variable. Finally, we print the dt_object to the console.

Note that the fromtimestamp() method assumes that the input timestamp value is in UTC. If the timestamp value is in a different timezone, you may need to adjust it accordingly before passing it to the method.

Rapt ‘N Rel

Python Documentation

time Module

The time module in Python provides functions for working with timestamps, which represent a point in time as the number of seconds elapsed since January 1, 1970 (also known as the Unix epoch). Here are some of the commonly used functions in the time module:

time()

This function returns the current timestamp in seconds since the epoch as a floating-point number.

```
import time


current_time = time.time()
print(current_time)
```

ctime()

This function takes a timestamp and returns a string representing the corresponding local time in a human-readable format.

```
import time


timestamp = time.time()
local_time = time.ctime(timestamp)
print(local_time)
```

gmtime()

This function takes a timestamp and returns a struct_time object representing the corresponding UTC time.

```
import time
```

```
timestamp = time.time()
utc_time = time.gmtime(timestamp)
print(utc_time)
```

strftime()

This function takes a struct_time object and a format string, and returns a string representing the time according to the format.

```
import time

timestamp = time.time()
local_time = time.localtime(timestamp)
formatted_time = time.strftime('%Y-%m-%d %H:%M:%S',
local_time)
print(formatted_time)
```

sleep()

This function suspends the execution of the current thread for a specified number of seconds.

```
import time

print('Starting...')
time.sleep(2)
print('Done!')
```

These are just a few examples of the functions available in the time module. The datetime module provides a more high-level interface for working with dates and times, which we covered in a previous question.

Rapt ‘N Rel

Python Documentation

time.sleep()

The time.sleep() function pauses the execution of a program for a specified number of seconds. It is often used in scenarios where you want to delay the execution of a task or simply add a delay between operations.

The time.sleep() function takes a single argument that specifies the number of seconds the program should wait before continuing with the execution. For example, the following code snippet will pause the execution of the program for 2 seconds:

```
import time


print("Hello")
time.sleep(2)
print("World")
```

In this example, the program will output "Hello", pause for 2 seconds using time.sleep(), and then output "World".

It is important to note that time.sleep() can have performance implications when used in long-running or high-performance applications. Pausing the execution of a program can also impact the responsiveness of a user interface, so it should be used judiciously.

Practice

Easy

Python programming practice questions that are straightforward.

Question 1

Write a Python program that accepts a string and prints the string reversed.

Solution:

```
string = input("Enter a string: ")
print(string[::-1])
```

Question 2

Write a Python program that accepts a number and checks if it is a prime number.

Solution:

```
num = int(input("Enter a number: "))

if num > 1:
    for i in range(2, num):
        if (num % i) == 0:
            print(num, "is not a prime number")
            break
    else:
        print(num, "is a prime number")
else:
    print(num, "is not a prime number")
```

Question 3

Write a Python program that accepts a list of numbers and returns the sum of all the even numbers.

Solution:

```
num_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

even_sum = 0
for num in num_list:
    if num % 2 == 0:
        even_sum += num

print("Sum of even numbers:", even_sum)
```

Question 4

Write a Python program that accepts a string and returns the number of vowels in the string.

Solution:

```
string = "Hello, World!"

vowels = "aeiouAEIOU"
count = 0

for char in string:
    if char in vowels:
        count += 1

print("Number of vowels:", count)
```

Question 5

Write a Python program to print the numbers from 1 to 10 using a for loop.

Solution:

```
for i in range(1, 11):
    print(i)
```

Question 6

Write a Python program that accepts a string and returns a new string with the first and last characters removed.

Solution:

```
string = "Hello, World!"

new_string = string[1:-1]

print("New string:", new_string)
```

Question 7

Write a Python program to check if a given number is even or odd.

Solution:

```
num = 4
if num % 2 == 0:
    print("Even")
else:
    print("Odd")
```

Rapt ‘N Rel

Python Documentation

Question 8

Write a Python program that accepts a string and checks if it is a palindrome.

Solution:

```
string = "racecar"

if string == string[::-1]:
    print("Palindrome")
else:
    print("Not a palindrome")
```

Question 9

Write a Python program to calculate the area of a rectangle with height 5 and width 10.

Solution:

```
height = 5
width = 10
area = height * width
print(area)
```

Question 10

Write a Python program that accepts a list of numbers and returns the average of all the numbers.

Solution:

```
num_list = [1, 2, 3, 4, 5]
```

```
average = sum(num_list) / len(num_list)

print("Average:", average)
```

Medium

Python programming practice questions that are of moderate difficulty.

Question 1

Write a Python program that accepts two lists and returns a list of elements common to both lists.

Solution:

```
list1 = [1, 2, 3, 4, 5]
list2 = [3, 4, 5, 6, 7]

common_elements = []

for element in list1:
    if element in list2 and element not in
common_elements:
        common_elements.append(element)

print("Common elements:", common_elements)
```

Rapt ‘N Rel

Python Documentation

Question 2

Write a Python program that accepts a list of strings and returns the longest string in the list.

Solution:

```
string_list = ["Hello", "World", "Python",
"Programming"]

longest_string = ""

for string in string_list:
    if len(string) > len(longest_string):
        longest_string = string

print("Longest string:", longest_string)
```

Question 3

Write a Python program to convert a temperature in Celsius to Fahrenheit. The formula is: F = C * 1.8 + 32

Solution:

```
celsius = 25
fahrenheit = celsius * 1.8 + 32
print(fahrenheit)
```

Question 4

Write a Python program to check if a given string is a palindrome.

Solution:

```
string = "racecar"
if string == string[::-1]:
    print("Palindrome")
else:
    print("Not a Palindrome")
```

Question 5

Write a Python program that accepts a list of numbers and returns a list of the squares of each number.

Solution:

```
num_list = [1, 2, 3, 4, 5]

squares = []

for num in num_list:
    squares.append(num**2)

print("Squares:", squares)
```

Question 6

Write a function that takes a string and returns the length of the longest palindromic substring in the string.

Solution:

```
def longest_palindromic_substring(string):
    max_len = 0
    start = 0
    for i in range(len(string)):
        for j in range(i, len(string)):
            if string[i:j+1] == string[i:j+1][::-1]:
                if j - i + 1 > max_len:
                    max_len = j - i + 1
                    start = i
    return max_len
```

Question 7

Write a function that takes a list of integers and returns a new list with all the subsets of the original list.

Solution:

```
def subsets(numbers):
    result = [[]]
    for n in numbers:
        result += [r + [n] for r in result]
    return result
```

Question 8

Write a function that takes a list of integers and returns the longest increasing subsequence in the list.

Solution:

```
def longest_increasing_subsequence(numbers):
    if not numbers:
        return []
    sequences = [[n] for n in numbers]
    for i in range(1, len(numbers)):
        for j in range(i):
            if numbers[j] < numbers[i]:
                if len(sequences[i]) <
len(sequences[j]) + 1:
                    sequences[i] = sequences[j] +
[numbers[i]]
    return max(sequences, key=len)
```

Question 9

Write a function that takes a list of integers and returns the product of all the numbers in the list except for the current number.

Solution:

```
def product_except_current(numbers):
    product = 1
    for n in numbers:
        product *= n
    return [product // n for n in numbers]
```

Rapt ‘N Rel

Python Documentation

Question 10

Write a function that takes two lists of integers and returns a new list with all the unique values in both lists.

Solution:

```
def unique_values(list1, list2):
    unique = set(list1 + list2)
    return list(unique)
```

Difficult

Python programming practice questions that are challenging.

Question 1

A company has a list of employees with their salaries, and they want to calculate the total salary expenditure for each department. Write a Python function that takes a list of employees with their salaries and department, and outputs a dictionary with each department's total salary expenditure.

Solution:

```
def
calculate_department_salary_expenditure(employees):
    department_salaries = {}
    for employee in employees:
        if employee['department'] in
department_salaries:
```

```
department_salaries[employee['department']] +=
employee['salary']
        else:

department_salaries[employee['department']] =
employee['salary']
    return department_salaries
```

Question 2

A school has a list of students with their grades, and they want to calculate the average grade for each class. Write a Python function that takes a list of students with their grades and class, and outputs a dictionary with each class's average grade.

Solution:

```
def calculate_class_average_grade(students):
    class_grades = {}
    for student in students:
        if student['class'] in class_grades:

class_grades[student['class']].append(student['grade'
])
        else:
            class_grades[student['class']] =
[student['grade']]
    class_average_grades = {}
    for class_name, grades in class_grades.items():
```

```
        class_average_grades[class_name] =
sum(grades) / len(grades)
    return class_average_grades
```

Question 3

A grocery store has a list of items with their prices and wants to calculate the total cost of a customer's shopping cart. Write a Python function that takes a list of items with their prices and quantities and outputs the total cost of the shopping cart.

Solution:

```
def calculate_shopping_cart_cost(items):
    total_cost = 0
    for item in items:
        total_cost += item['price'] *
item['quantity']
    return total_cost
```

Question 4

A company wants to assign tasks to employees based on their skills and workload. Write a Python function that takes a list of employees, their skills, and their current workload, and outputs a list of recommended tasks for each employee based on their skills and availability.

Solution:

```
def assign_tasks_to_employees(employees, tasks):
    for task in tasks:
        eligible_employees = []
```

```
        for employee in employees:
            if task['skill'] in employee['skills']
and employee['workload'] < 1.0:
                eligible_employees.append(employee)
        if eligible_employees:
            selected_employee =
min(eligible_employees, key=lambda e: e['workload'])

selected_employee['tasks'].append(task['name'])
            selected_employee['workload'] +=
task['time_required']
    return employees
```

Question 5

A hospital has a list of patients with their medical conditions and wants to assign nurses to patients based on their medical conditions. Write a Python function that takes a list of patients with their medical conditions and a list of available nurses with their specialties, and outputs a dictionary with each patient's assigned nurse.

Solution:

```
def assign_nurses_to_patients(patients, nurses):
    patient_nurse_dict = {}
    for patient in patients:
        eligible_nurses = []
        for nurse in nurses:
            if patient['condition'] in
nurse['specialties'] and nurse['is_available']:
                eligible_nurses.append(nurse)
```

```
        if eligible_nurses:
            selected_nurse = min(eligible_nurses,
key=lambda n: len(n['assigned_patients']))

selected_nurse['assigned_patients'].append(patient['n
ame'])
            patient_nurse_dict[patient['name']] =
selected_nurse['name']
    return patient_nurse_dict
```

Very Difficult

Python programming practice questions that are highly challenging

Question 1

You are tasked with building a cryptocurrency trading bot that can predict the value of Bitcoin and other cryptocurrencies accurately. The bot must be able to make trades based on its predictions, and it must be able to learn from its trades.

Write an object-oriented Python program that **implements** this trading bot, with the following features:

- The bot should use machine learning algorithms to predict the value of cryptocurrencies based on historical data.
- The bot should have a trading strategy that is based on these predictions, and it should be able to execute trades on a cryptocurrency exchange API.
- The bot should be able to learn from its trades and adjust its trading strategy accordingly.
- The bot should be able to handle multiple cryptocurrencies and multiple exchanges.

Rapt ‘N Rel

Python Documentation

NB: Create the OOP structure, you’re not required to make the actual bot.

Solution:

```
# Create a class for the trading bot:
class TradingBot:
    def __init__(self):
        # Initialize the trading bot
        pass

    def train(self):
        # Train the trading bot using machine
learning algorithms
        pass

    def predict(self):
        # Use the trained model to predict the value
of cryptocurrencies
        pass

    def trade(self):
        # Execute trades on a cryptocurrency exchange
API based on the predictions
        pass

    def learn(self):
        # Learn from the trades and adjust the
trading strategy accordingly
```

```
        pass

# Initialize the trading bot:
bot = TradingBot()

# Train the trading bot using historical data:
bot.train()

# Predict the value of cryptocurrencies:
prediction = bot.predict()

#Execute trades based on the predictions:
bot.trade()

# Learn from the trades and adjust the trading
strategy accordingly:
bot.learn()
```

Question 2

A company wants to build a system to manage their employees' data. Each employee has a name, an ID, and a salary. The system should allow adding, removing, and updating employees' information, as well as calculating the total payroll. Implement the system using Python OOP.

Solution:

```
class Employee:
    def __init__(self, name, id, salary):
        self.name = name
```

```
        self.id = id
        self.salary = salary

class PayrollSystem:
    def __init__(self):
        self.employees = []

    def add_employee(self, employee):
        self.employees.append(employee)

    def remove_employee(self, employee):
        self.employees.remove(employee)

    def update_employee(self, employee, name, id,
salary):
        employee.name = name
        employee.id = id
        employee.salary = salary

    def calculate_payroll(self):
        total_payroll = 0
        for employee in self.employees:
            total_payroll += employee.salary
        return total_payroll
```

Rapt ‘N Rel

Python Documentation

Question 3

A school wants to track the attendance of its students. Each student has a name, an ID, and a list of classes. The attendance system should allow adding and removing students, marking their attendance for each class, and generating a report of attendance rates for each class. Implement the attendance system using Python OOP.

Solution:

```
class Student:
    def __init__(self, name, id, classes):
        self.name = name
        self.id = id
        self.classes = classes
        self.attendance = {}

    def mark_attendance(self, class_name,
attendance):
        self.attendance[class_name] = attendance

class AttendanceSystem:
    def __init__(self):
        self.students = []

    def add_student(self, student):
        self.students.append(student)

    def remove_student(self, student):
        self.students.remove(student)
```

```
    def generate_report(self, class_name):
        total_students = 0
        present_students = 0
        for student in self.students:
            if class_name in student.classes:
                total_students += 1
                if class_name in student.attendance
and student.attendance[class_name]:
                    present_students += 1
        attendance_rate = present_students /
total_students if total_students > 0 else 0
        return attendance_rate
```

Question 4

You are tasked with building a simulation of a self-driving car system. The system should be able to navigate a complex road network, and safely transport passengers from one location to another. The car has sensors to detect obstacles and other cars, as well as a GPS system to navigate to a destination.

Implement an object-oriented Python program that simulates this self-driving car system, with the following features:

- The car should be able to navigate a complex road network, including highways, city streets, and rural roads.
- The car should be able to detect obstacles and other cars, and adjust its speed and direction accordingly to avoid collisions.

Solution:

Rapt ‘N Rel

Python Documentation

```
import random

class Road:
    def __init__(self, start, end, length):
        self.start = start
        self.end = end
        self.length = length
        self.vehicles = []

class Intersection:
    def __init__(self):
        self.roads = []

class Car:
    MAX_SPEED = 60 # miles per hour
    MIN_DISTANCE = 50 # feet

    def __init__(self, start_road, start_position,
destination_road):
        self.current_road = start_road
        self.current_position = start_position
        self.destination_road = destination_road
        self.speed = 0

    def drive(self):
        if self.current_road ==
self.destination_road:
```

```
            return # already at destination
        if self.speed < self.MAX_SPEED:
            self.speed += 5 # accelerate
        if self.speed >
self.current_road.speed_limit:
            self.speed =
self.current_road.speed_limit # slow down
        if self.speed > self.distance_to_next_car():
            self.speed = self.distance_to_next_car()
- self.MIN_DISTANCE # slow down to avoid collision
        if self.current_position + self.speed >=
self.current_road.length:
            # reached end of current road
            next_intersection = self.current_road.end
            possible_roads = [r for r in
next_intersection.roads if r != self.current_road]
            self.current_road =
random.choice(possible_roads) # choose a random road
            self.current_position = 0 # start at the
beginning of the new road
        else:
            # continue on current road
            self.current_position += self.speed

    def distance_to_next_car(self):
        for vehicle in self.current_road.vehicles:
```

```
            if vehicle.current_position >
self.current_position:
                return vehicle.current_position -
self.current_position
        return float("inf")

class SelfDrivingCar:
    def __init__(self):
        self.roads = []
        self.intersections = []
        self.cars = []

    def add_road(self, start, end, length,
speed_limit):
        road = Road(start, end, length)
        road.speed_limit = speed_limit
        self.roads.append(road)
        start.roads.append(road)

    def add_intersection(self):
        intersection = Intersection()
        self.intersections.append(intersection)

    def add_car(self, start_road, start_position,
destination_road):
        car = Car(start_road, start_position,
destination_road)
```

```
            self.cars.append(car)
            start_road.vehicles.append(car)

        def drive_cars(self):
            for car in self.cars:
                car.drive()

        def display_map(self):
            for road in self.roads:
                print(f"{road.start} -> {road.end}:
{road.length} feet, {road.speed_limit} mph")

        def display_cars(self):
            for car in self.cars:
                print(f"Car at {car.current_road.start}
position {car.current_position}, speed {car.speed}")
```

Question 5

Based on Question 4 show how the program can be used

Solution:

```
# Create a self-driving car system:
car_system = SelfDrivingCar()

# Add roads to the system:
road1 = car_system.add_road("A", "B", 5000, 40)
```

```
road2 = car_system.add_road("B", "C", 3000, 50)
road3 = car_system.add_road("B", "D", 2000, 60)
road4 = car_system.add_road
```

Interview Questions

Common

Frequently encountered programming interview questions.

Question 1

Write a Python program to find the largest element in a list.

Solution:

```
def find_largest_element(lst):
    max_element = lst[0]
    for i in range(1, len(lst)):
        if lst[i] > max_element:
            max_element = lst[i]
    return max_element

# Example usage
```

```
lst = [10, 20, 30, 40, 50]
print(find_largest_element(lst)) # Output: 50
```

Question 2

Write a Python program to check if a string is a palindrome.

Solution:

```
def is_palindrome(s):
    return s == s[::-1]

# Example usage
s = "racecar"
print(is_palindrome(s)) # Output: True
```

Question 3

Write a Python program to reverse a string.

Solution:

```
def reverse_string(s):
    return s[::-1]

# Example usage
s = "hello world"
print(reverse_string(s)) # Output: "dlrow olleh"
```

Rapt ‘N Rel

Python Documentation

Question 4

Write a Python program to sort a list of integers in ascending order.

Solution:

```
def sort_list(lst):
    return sorted(lst)

# Example usage
lst = [3, 1, 4, 2, 5]
print(sort_list(lst)) # Output: [1, 2, 3, 4, 5]
```

Question 5

Write a Python program to calculate the factorial of a number.

Solution:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

# Example usage
n = 5
print(factorial(n)) # Output: 120
```

Rapt ‘N Rel

Python Documentation

Question 6

Write a Python program to find the second largest element in a list.

Solution:

```
def find_second_largest(lst):
    max_element = max(lst)
    lst.remove(max_element)
    second_max = max(lst)
    return second_max

# Example usage
lst = [10, 20, 30, 40, 50]
print(find_second_largest(lst)) # Output: 40
```

Question 7

Write a Python program to find the sum of all elements in a list.

Solution:

```
def sum_list(lst):
    sum = 0
    for i in lst:
        sum += i
    return sum

# Example usage
lst = [1, 2, 3, 4, 5]
```

```
print(sum_list(lst)) # Output: 15
```

Question 8

Write a Python program to check if a number is prime.

Solution:

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

# Example usage
n = 7
print(is_prime(n)) # Output: True
```

Question 9

Write a Python program to find the intersection of two lists.

Solution:

```
def intersection(lst1, lst2):
    return list(set(lst1) & set(lst2))

# Example usage
```

```
lst1 = [1, 2, 3, 4, 5]
lst2 = [3, 4, 5, 6, 7]
print(intersection(lst1, lst2)) # Output: [3, 4, 5]
```

Question 10

Write a Python program to count the number of vowels in a string.

Solution:

```
def count_vowels(s):
    vowels = "aeiou"
    count = 0
    for char in s:
        if char.lower() in vowels:
            count += 1
    return count

# Example usage
s = "hello world"
print(count_vowels(s)) # Output: 3
```

Rapt ‘N Rel

Python Documentation

Tough

Challenging programming interview questions experienced by candidates.

Question 1

Write a Python program to check if a given string is a palindrome.

Solution:

```
def is_palindrome(s):
    s = s.lower().replace(" ", "")
    return s == s[::-1]

# Example usage
s = "A man a plan a canal Panama"
print(is_palindrome(s)) # Output: True
```

Question 2

Write a Python program to find the maximum sum of a contiguous subarray within a given list of integers.

Solution:

```
def max_subarray_sum(lst):
    max_so_far = lst[0]
    max_ending_here = lst[0]
    for i in range(1, len(lst)):
```

```
        max_ending_here = max(lst[i], max_ending_here
+ lst[i])
        max_so_far = max(max_so_far, max_ending_here)
    return max_so_far

# Example usage
lst = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print(max_subarray_sum(lst)) # Output: 6
```

Question 3

Write a Python program to implement binary search.

Solution:

```
def binary_search(lst, x):
    low, high = 0, len(lst) - 1
    while low <= high:
        mid = (low + high) // 2
        if lst[mid] < x:
            low = mid + 1
        elif lst[mid] > x:
            high = mid - 1
        else:
            return mid
    return -1

# Example usage
```

```
lst = [2, 3, 4, 10, 40]
x = 10
print(binary_search(lst, x)) # Output: 3
```

Question 4

Write a Python program to find the longest common prefix among a list of strings.

Solution:

```
def longest_common_prefix(strs):
    if not strs:
        return ""
    prefix = strs[0]
    for s in strs[1:]:
        while prefix and s[:len(prefix)] != prefix:
            prefix = prefix[:-1]
        if not prefix:
            return ""
    return prefix

# Example usage
strs = ["flower", "flow", "flight"]
print(longest_common_prefix(strs)) # Output: "fl"
```

Question 5

Write a Python program to reverse a linked list.

Rapt ‘N Rel

Python Documentation

Solution:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def add(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            curr = self.head
            while curr.next:
                curr = curr.next
            curr.next = new_node

    def reverse(self):
        prev = None
        curr = self.head
        while curr:
            next = curr.next
            curr.next = prev
            prev = curr
```

```
            curr = next
        self.head = prev

    def display(self):
        curr = self.head
        while curr:
            print(curr.data)
            curr = curr.next

# Example usage
ll = LinkedList()
ll.add(1)
ll.add(2)
ll.add(3)
print("Original:")
ll.display()
ll.reverse()
print("Reversed:")
ll.display()
```

Question 6

Write a function to determine if a string is a palindrome (reads the same forward and backward).

Solution:

```
def is_palindrome(string):
    return string == string[::-1]
```

Rapt ‘N Rel

Python Documentation

Question 7

Given a list of integers, write a function to find the maximum sum of any contiguous subarray of the list.

Solution:

```
def max_subarray_sum(nums):
    max_sum = float('-inf')
    curr_sum = 0
    for num in nums:
        curr_sum = max(num, curr_sum + num)
        max_sum = max(max_sum, curr_sum)
    return max_sum
```

Question 8

Write a function to find the first non-repeating character in a string.

Solution:

```
from collections import Counter

def first_non_repeating(string):
    counter = Counter(string)
    for char in string:
        if counter[char] == 1:
            return char
    return None
```

Rapt ‘N Rel

Python Documentation

Question 9

Given a list of intervals (as tuples), write a function to merge any overlapping intervals.

Solution:

```
def merge_intervals(intervals):
    intervals = sorted(intervals)
    merged = []
    for interval in intervals:
        if not merged or merged[-1][1] < interval[0]:
            merged.append(interval)
        else:
            merged[-1] = (merged[-1][0], max(merged[-
1][1], interval[1]))
    return merged
```

Question 10

Write a function to determine if a given string is a valid IPv4 address.

Solution:

```
def is_valid_IPv4(address):
    parts = address.split('.')
    if len(parts) != 4:
        return False
    for part in parts:
        if not part.isdigit() or int(part) < 0 or
int(part) > 255:
```

```
            return False
        if len(part) > 1 and part[0] == '0':
            return False
    return True
```

Missed

Commonly failed programming interview questions

Question 1

Write a function to reverse a linked list.

Solution:

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverse_linked_list(head):
    prev = None
    curr = head
    while curr:
        next_node = curr.next
        curr.next = prev
        prev = curr
        curr = next_node
    return prev
```

Rapt ‘N Rel

Python Documentation

Question 2

Given a list of integers and a target sum, write a function to find all pairs of integers in the list that add up to the target sum.

Solution:

```
def two_sum(nums, target):
    seen = set()
    pairs = set()
    for num in nums:
        complement = target - num
        if complement in seen:
            pairs.add((min(num, complement), max(num,
complement)))
        seen.add(num)
    return pairs
```

Question 3

Write a function to sort a list of integers in ascending order using the quicksort algorithm.

Solution:

```
def quicksort(nums):
    if len(nums) <= 1:
        return nums
    pivot = nums[len(nums)//2]
    left = [num for num in nums if num < pivot]
```

```
    middle = [num for num in nums if num == pivot]
    right = [num for num in nums if num > pivot]
    return quicksort(left) + middle +
quicksort(right)
```

Question 4

Given a string representing a mathematical expression, write a function to evaluate the expression and return the result.

Solution:

```
def evaluate_expression(expression):
    stack = []
    num = 0
    operator = '+'
    for i in range(len(expression)):
        if expression[i].isdigit():
            num = num * 10 + int(expression[i])
        if not expression[i].isdigit() and not
expression[i].isspace() or i == len(expression) - 1:
            if operator == '+':
                stack.append(num)
            elif operator == '-':
                stack.append(-num)
            elif operator == '*':
                stack.append(stack.pop() * num)
            elif operator == '/':
                stack.append(int(stack.pop() / num))
```

```
            operator = expression[i]
            num = 0
    return sum(stack)
```

Question 5

Write a function to find the kth largest element in a list of integers.

Solution:

```
def find_kth_largest(nums, k):
    return sorted(nums, reverse=True)[k-1]
```

Question 6

Write a function to swap two variables without using a temporary variable.

Solution:

```
def swap(a, b):
    a = a + b
    b = a - b
    a = a - b
    return a, b
```

Question 7

Given a list of integers, write a function to find the median value.

Solution:

```
def find_median(nums):
    nums.sort()
    n = len(nums)
    if n % 2 == 0:
        return (nums[n // 2] + nums[n // 2 - 1]) / 2
    else:
        return nums[n // 2]
```

Question 8

Write a function to determine if a given string is a palindrome, ignoring non-alphanumeric characters and cases.

Solution:

```
def is_palindrome(string):
    left = 0
    right = len(string) - 1
    while left < right:
        while left < right and not
string[left].isalnum():
            left += 1
        while left < right and not
string[right].isalnum():
            right -= 1
        if string[left].lower() !=
string[right].lower():
            return False
        left += 1
```

```
            right -= 1
        return True
```

Question 9

Write a function to remove all duplicates from a list, preserving the order of the elements.

Solution:

```
def remove_duplicates(nums):
    seen = set()
    result = []
    for num in nums:
        if num not in seen:
            seen.add(num)
            result.append(num)
    return result
```

Question 10

Given two sorted arrays, write a function to merge them into a single sorted array.

Solution:

```
def merge_sorted_arrays(arr1, arr2):
    merged_array = []
    i, j = 0, 0
    while i < len(arr1) and j < len(arr2):
        if arr1[i] < arr2[j]:
            merged_array.append(arr1[i])
```

```
            i += 1
        else:
            merged_array.append(arr2[j])
            j += 1
    while i < len(arr1):
        merged_array.append(arr1[i])
        i += 1
    while j < len(arr2):
        merged_array.append(arr2[j])
        j += 1
    return merged_array
```