Chapter 1: Introduction to Python

History of Python

Python is a high-level, interpreted programming language that was first released in 1991. It was created by Guido van Rossum, a Dutch computer scientist, while working at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Van Rossum named the language after the British comedy group Monty Python, and intended for it to be a fun and easy-to-use language for beginners. Python was designed with the philosophy of "There's only one way to do it" and "Readability counts," which emphasizes simplicity and clarity in code.

Python quickly gained popularity for its simplicity, readability, and extensive standard library, which made it easy for programmers to perform common tasks without having to write their own code. It also has a large and active community of developers, which has contributed to its continued growth and evolution.

Over the years, Python has been used in a variety of fields, including web development, scientific computing, data analysis, and artificial intelligence. It is now one of the most popular programming languages in the world, and is widely used in industries such as finance, education, and government.

Key features of Python

Python is known for its simplicity, readability, and flexibility, as well as a number of other key features that make it an attractive programming language for beginners and experienced programmers alike. Some of the key features of Python include:

1. Interpreted language: Python is an interpreted language, which means that the code is executed line by line, as opposed to being compiled all at once. This makes it easy to debug and test code, as well as allowing for interactive use.
2. Dynamic typing: Python uses dynamic typing, which means that variables do not have to be explicitly declared with a data type. This allows for greater flexibility and reduces the amount of code needed.
3. High-level language: Python is a high-level language, which means that it is abstracted from the details of the computer's hardware and operating system. This makes it easier to write and understand code, as well as making it easier to port between different platforms.
4. Extensive standard library: Python comes with a large and comprehensive standard library, which includes modules for tasks such as connecting to web servers, reading and writing files, and working with data.
5. Large and active community: Python has a large and active community of developers, which has contributed to the language's continued growth and evolution. There are also many resources available for learning Python,

including online tutorials, books, and forums.
6. Object-oriented programming: Python supports object-oriented programming, which allows for the creation of reusable code through the use of classes and objects.
7. Versatility: Python is a versatile language that can be used for a wide range of tasks, including web development, scientific computing, data analysis, and artificial intelligence.

Setting up a Python development environment

To set up a Python development environment, you will need to install a Python interpreter and a text editor or integrated development environment (IDE). Here are the steps to set up a basic Python development environment:

1. Install Python: The first step is to install a Python interpreter on your computer. You can download the latest version of Python from the official website (**https://www.python.org/downloads/**). Make sure to select the appropriate version for your operating system (e.g., Windows, macOS, Linux).
2. Choose a text editor or IDE: Next, you will need a text editor or IDE to write and edit your Python code. Some popular options include IDLE (included with Python), PyCharm, and Visual Studio Code.
3. Create a Python file: Once you have a text editor or IDE installed, you can create a new Python file by selecting "New File" or "Create New File" from the File menu. You can then write your Python code in this file.
4. Run your code: To run your Python code, you can either use the Run command in your text editor or IDE, or you can use the Python interpreter to run the code from the command line. To do this, open a terminal or command prompt and navigate to the directory where your Python file is located. Then, enter the command "python [filename]" to run the code.
5. Debug your code: If you encounter any errors or bugs in your code, you can use the debugger built into your text editor or IDE to troubleshoot the issue. You can also use the "print()" function to output values and debug your code.

By following these steps, you should have a basic Python development environment set up and be ready to start writing and running Python code.

Basic syntax and data types

Basic Syntax

There are several basic syntax rules and data types that you should be familiar with. Here are some key points to remember:

1. Indentation: Indentation is used to indicate blocks of code. For example, a block of code within an if statement or a for loop should be indented by four spaces (or one tab).

2. Comments: You can add comments to your code by using the "#" symbol. Anything after the "#" symbol on a line will be treated as a comment and ignored by the interpreter.
3. Data types: Python has several built-in data types, including integers, floating-point numbers, strings, and booleans. You can use these data types to store and manipulate data in your code.
4. Variables: You can use variables to store and manipulate data. To create a variable, you simply assign a value to a name. For example, to create a variable called "x" and assign it the value 5, you would write "x = 5".
5. Operators: Python has several operators that you can use to perform operations on data, including arithmetic operators (+, -, *, /), comparison operators (==, !=, >, <, >=, <=), and logical operators (and, or, not).
6. Control structures: Python has several control structures that you can use to control the flow of your code, including if statements, for loops, and while loops.
7. Functions: You can define your own functions to organize and reuse your code. To create a function, you use the "def" keyword, followed by the function name and any necessary arguments.
8. Modules: Python has a large standard library of modules that you can use to perform common tasks, such as reading and writing files, connecting to the internet, and working with dates and times. You can also import and use third-party modules to extend the functionality of your code.
9. Exception handling: You can use try-except statements to handle errors and exceptions that may occur in your code. This allows you to write code that can gracefully handle unexpected situations.
10. Object-oriented programming: Python is an object-oriented programming language, which means that you can use it to create classes and objects to represent real-world concepts in your code.

By learning these concepts and using them in your code, you will be able to write powerful and efficient Python programs.

Data Types

There are several built-in data types that you can use to store and manipulate data. These data types include:

1. Integers
2. Float
3. Strings
4. Booleans
5. List
6. Tuple
7. Set
8. Dictionary
9. Frozen set
10. Bytes
11. Byte Array

Integers

Integers are whole numbers that can be positive, negative, or zero. For example, 0, 1, -2, and 100 are all integers. You can use the int data type to store integers. Here is an example:

```
# Declaring and using an integer
x = 10
print(x)  # Output: 10
1, 2
# To check the type

print(type(x))  # Output: <class 'int'>

# Performing arithmetic operations with integers
y = 20
z = x + y
print(z)  # Output: 30

# Dividing integers
a = 10
b = 3
c = a / b
print(c)  # Output: 3.3333333333333335

# Dividing integers using floor division
d = a // b
print(d)  # Output: 3

# Getting the remainder of a division operation
e = a % b
print(e)  # Output: 1

# Exponentiation
f = a ** b
print(f)  # Output: 1000
```

Floats

Floats are numbers with decimal points. For example, 0.5, 3.14, and -1.0 are all floats. You can use the float data type to store floats.

```
# Declaring and using a float
x = 10.5
print(x)  # Output: 10.5

# To check the type
print(type(x))  # Output: <class 'float'>
```

```
# Performing arithmetic operations with floats
y = 20.7
z = x + y
print(z)  # Output: 31.2

# Dividing floats
a = 10.5
b = 3.1
c = a / b
print(c)  # Output: 3.387096774193548

# Getting the remainder of a division operation
# Note that this will not work with floats as it does not give accurate results
d = a % b
print(d)  # Output: TypeError: can't mod floats

# Exponentiation
e = a ** b
print(e)  # Output: 587.5974286844106
```

Strings

Strings are sequences of characters, such as words or sentences. For example, "hello", "python", and "The quick brown fox jumps over the lazy dog" are all strings. You can use the str data type to store strings.

```
# Declaring and using a string
x = "hello"
print(x)  # Output: "hello"

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

# Getting the length of a string
a = len(x)
print(a)  # Output: 5

# Accessing individual characters in a string
b = x[0]  # First character
c = x[-1]  # Last character
print(b)  # Output: "h"
print(c)  # Output: "o"

# Slicing a string to get a portion of it
```

```
d = x[1:3]  # Characters at index 1 and 2 (but not 3)
print(d)  # Output: "el"

# Repeating a string multiple times
e = x * 3
print(e)  # Output: "hellohellohello"

# Checking if a string is present in another string
f = "he" in x
g = "hi" in x
print(f)  # Output: True
print(g)  # Output: False

# Changing the case of a string
h = x.upper()
i = x.lower()
print(h)  # Output: "HELLO"
print(i)  # Output: "hello"
```

Booleans

Booleans are values that represent true or false. For example, True and False are both booleans. You can use the bool data type to store booleans.

```
# Assign a boolean value to a variable
flag = True

# Check if a variable is a boolean
is_bool = isinstance(flag, bool)
print(is_bool)  # Output: True

# Use a boolean in a conditional statement
if flag:
    print("Flag is True")
else:
    print("Flag is False")  # Output: "Flag is True"

# Use the and operator
if flag and is_bool:
    print("Both values are True")  # Output: "Both values are True"

# Use the or operator
if flag or not is_bool:
    print("At least one value is True")  # Output: "At least one value is True"
```

Lists:

Lists are ordered collections of items that can be of any data type. For example,

[1, 2, 3] is a list of integers, while [“apple”, “banana”, “cherry”] is a list of strings. You can use the list data type to store lists.

```
# Create a list of integers
numbers = [1, 2, 3, 4, 5]

# Create a list of strings
words = ['apple', 'banana', 'cherry']

# Create a list of mixed data types
mixed = [1, 'apple', 3.14, True]

# Access an item in the list by its index
print(words[0])  # Output: 'apple'

# Modify an item in the list
words[1] = 'pear'
print(words)  # Output: ['apple', 'pear', 'cherry']

# Check the length of a list
length = len(numbers)
print(length)  # Output: 5

# Loop through a list
for item in words:
    print(item)  # Output: 'apple', 'pear', 'cherry'

# Check if an item is in a list
if 'apple' in words:
    print("Apple is in the list")  # Output: "Apple is in the list"
```

Lists also have various built-in methods for adding, removing, and manipulating items, such as append, insert, remove, pop, and sort. Here is an example:

```
# Add an item to the end of the list
words.append('mango')
print(words)  # Output: ['apple', 'pear', 'cherry', 'mango']

# Insert an item at a specific index
words.insert(1, 'orange')
print(words)  # Output: ['apple', 'orange', 'pear', 'cherry', 'mango']

# Remove an item from the list
words.remove('pear')
print(words)  # Output: ['apple', 'orange', 'cherry', 'mango']

# Remove the last item from the list
```

```
last = words.pop()
print(last)  # Output: 'mango'
print(words)  # Output: ['apple', 'orange', 'cherry']

# Sort the items in the list
words.sort()
print(words)  # Output: ['apple', 'cherry', 'orange']
```

Tuples:

Tuples are immutable sequences of objects. This means that once a tuple is created, you cannot change its values. Tuples are created by enclosing a comma-separated sequence of values in parentheses. For example:

```
>>> t = (1, 2, 3)
>>> type(t)
<class 'tuple'>
```

You can access the elements of a tuple by using indexing, just like you would with a list. For example:

```
>>> t = (1, 2, 3)
>>> t[0]
1
>>> t[1]
2
>>> t[2]
3
```

You can also use slicing to access a range of elements within a tuple. For example:

```
>>> t = (1, 2, 3, 4, 5)
>>> t[1:4]
(2, 3, 4)
```

Tuples also support all of the common sequence operations, such as concatenation, repetition, and membership testing. For example:

```
>>> t1 = (1, 2, 3)
>>> t2 = (4, 5, 6)
>>> t1 + t2
(1, 2, 3, 4, 5, 6)
>>> t1 * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> 2 in t1
True
```

You can also use the built-in function `len()` to get the length of a tuple. For example:

```
>>> t = (1, 2, 3)
>>> len(t)
3
```

Tuples are often used to store related pieces of information, such as the name and age of a person. For example:

```
>>> person = ("Alice", 25)
>>> name, age = person
>>> name
"Alice"
>>> age
25
```

You can also use tuples to return multiple values from a function. For example:

```
def divide(x, y):
    quotient = x // y
    remainder = x % y
    return quotient, remainder

>>> divide(10, 3
(3, 1)
```

Dictionaries:

Dictionaries are unordered collections of key-value pairs. It is a mutable data type, which means that you can change the contents of a dictionary after it is created.

Here is an example of creating a dictionary:

```
# create a dictionary with key-value pairs
my_dict = {'name': 'John', 'age': 25, 'location': 'New York'}

# access a value in the dictionary using its key
print(my_dict['name'])  # Output: 'John'

# change the value of a key
my_dict['age'] = 26
print(my_dict['age'])  # Output: 26

# add a new key-value pair to the dictionary
my_dict['gender'] = 'male'
print(my_dict)  # Output: {'name': 'John', 'age': 26, 'location': 'New York', 'gender': 'ma

# delete a key-value pair from the dictionary
del my_dict['location']
print(my_dict)  # Output: {'name': 'John', 'age': 26, 'gender': 'male'}
```

You can also create a dictionary using the built-in function **dict()**. Here is an example:

```
# create a dictionary using the dict() function
my_dict = dict(name='John', age=25, location='New York')
print(my_dict)  # Output: {'name': 'John', 'age': 25, 'location': 'New York'}
```

You can also create a dictionary from two lists using the built-in function **zip()**. Here is an example:

```
# create two lists
keys = ['name', 'age', 'location']
values = ['John', 25, 'New York']

# create a dictionary using zip()
my_dict = dict(zip(keys, values))
print(my_dict)  # Output: {'name': 'John', 'age': 25, 'location': 'New York'}
```

Sets:

Sets are unordered collections of unique elements. They are unordered and do not allow duplicate values. Sets are created using curly braces {} or the set() function.

```
# Creating a set
numbers = {1, 2, 3, 4, 5}

# Checking the type of the set
print(type(numbers))  # Output: <class 'set'>

# Adding an element to the set
numbers.add(6)

# Removing an element from the set
numbers.remove(2)

# Checking if an element is in the set
print(1 in numbers)  # Output: True
print(2 in numbers)  # Output: False

# Iterating through the set
for num in numbers:
  print(num)  # Output: 3, 4, 5, 6

# Set operations
evens = {2, 4, 6, 8}
odds = {1, 3, 5, 7}

# Union of sets
```

```
all_numbers = evens.union(odds)
print(all_numbers)  # Output: {2, 4, 6, 8, 1, 3, 5, 7}

# Intersection of sets
even_odds = evens.intersection(odds)
print(even_odds)  # Output: set()

# Difference between sets
only_evens = evens.difference(odds)
print(only_evens)  # Output: {2, 4, 6, 8}

# Clear the set
evens.clear()
print(evens)  # Output: set()
```

Frozen Sets:

Frozen sets are similar to sets, but they are immutable, meaning that they cannot be modified once created. They are created using the **`frozenset()`**function, and they can be used to store an unordered collection of unique elements.

Frozen sets are useful when you want to store a set of elements that you don't want to modify, for example, if you want to use a set as a key in a dictionary. They are also faster than sets in some cases, because they are implemented using a hash table and do not require the overhead of resizing and rehashing when elements are added or removed.

```
# create a frozen set
frozen_set = frozenset([1, 2, 3, 4])

# print the frozen set
print(frozen_set)  # Output: frozenset({1, 2, 3, 4})

# try to add an element to the frozen set
frozen_set.add(5)  # This will raise an AttributeError

# try to remove an element from the frozen set
frozen_set.remove(4)  # This will raise an AttributeError
```

Bytes:

The **`bytes`** data type represents a sequence of values that represent ASCII characters. It is similar to a list of integers, but each element must be in the range 0-255. The **`bytes`** data type is immutable, meaning that once it is created, the elements cannot be changed.

Here is an example of creating and accessing elements of a **`bytes`** object:

```
# Create bytes object with values 65, 66, 67, which
# represent ASCII characters 'A', 'B', and 'C'
```

```
b = bytes([65, 66, 67])

# Access the first element of the bytes object
print(b[0])  # Output: 65

# Access the last element of the bytes object
print(b[-1])  # Output: 67

# Access a range of elements in the bytes object
print(b[1:3])  # Output: b'BC'

# Iterate over the elements of the bytes object
for element in b:
    print(element)  # Output: 65 66 67
```

It is also possible to create a **bytes**object from a string using the **encode()**method. The string must be ASCII encoded, meaning that it can only contain characters that have an ASCII representation.

```
# Create a bytes object from a string using the encode() method
b = "Hello, World!".encode('ascii')

print(b)  # Output: b'Hello, World!'
```

Byte Arrays:

Byte arrays are a mutable sequence type, similar to a list of integers but with the additional ability to store and manipulate binary data. They are often used for reading and writing files, or for storing and manipulating raw data such as image or audio data.

```
# Create a byte array with three elements
b = bytearray(3)

# Set the values of the elements
b[0] = 65
b[1] = 66
b[2] = 67

print(b)  # Output: bytearray(b'ABC')
```

Complex Numbers:

Complex numbers are a type of data that represents numbers with a real and imaginary component. They are denoted by a real part followed by a "j" representing the imaginary component. For example, 3+4j is a complex number with a real part of 3 and an imaginary part of 4.

You can use the complex() function to create complex numbers. For example:

```
# Create a complex number with a real part of 3 and an imaginary part of 4
complex_number = complex(3, 4)
print(complex_number)  # Output: (3+4j)

# Create a complex number with a real part of 2.5 and an imaginary part of -1
complex_number = complex(2.5, -1)
print(complex_number)  # Output: (2.5-1j)

# Create a complex number with a real part of 0 and an imaginary part of 1
complex_number = complex(0, 1)
print(complex_number)  # Output: 1j
```

data:image/svg+xml,%3csvg%20xmlns=%27http://www.w3.org/2000/svg%27%20version=%271.1%27%20width=%2730%27%20height=%2730%27/%3e

You can also perform arithmetic operations on complex numbers just like you would with regular numbers. For example:

```
# Add two complex numbers
complex_number_1 = 3+4j
complex_number_2 = 1+2j
result = complex_number_1 + complex_number_2
print(result)  # Output: (4+6j)

# Subtract two complex numbers
complex_number_1 = 3+4j
complex_number_2 = 1+2j
result = complex_number_1 - complex_number_2
print(result)  # Output: (2+2j)

# Multiply two complex numbers
complex_number_1 = 3+4j
complex_number_2 = 1+2j
result = complex_number_1 * complex_number_2
print(result)  # Output: (-5+10j)

# Divide two complex numbers
complex_number_1 = 3+4j
complex_number_2 = 1+2j
result = complex_number_1 / complex_number_2
print(result)  # Output: (1.6+0.4j)
```

You can also use the built-in functions abs(), real(), and imag() to get the absolute value, real part, and imaginary part of a complex number, respectively. For example:

```
complex_number = 3+4j
```

```
# Get the absolute value of a complex number (distance from the origin in the complex plane)
abs_val = abs(complex_number)
print(abs_val)  # Output: 5.0

# Get the real part of a complex number
real_part = complex_number.real
print(real_part)  # Output: 3.0

# Get the imaginary part of a complex number
imag_part = complex_number.imag
print(imag_part)  # Output: 4.0
```

Variables and operations

Variables are used to store values that can be manipulated or used in various ways. They are created simply by assigning a value to a name.

```
x = 10
y = 20

# Reassigning values to variables
x = 30
y = 40

# Using variables in expressions
z = x + y
print(z)  # Output: 70
```

Python supports various types of operations such as

- arithmetic,
- comparison,
- logical, and
- bitwise.

Here are a few examples:

Arithmetic operations:

```
x = 10
y = 20

# Addition
print(x + y)  # Output: 30

# Subtraction
print(x - y)  # Output: -10

# Multiplication
```

```
print(x * y)  # Output: 200

# Division
print(x / y)  # Output: 0.5

# Modulus (remainder)
print(x % y)  # Output: 10

# Exponentiation
print(x ** y)  # Output: 100000000000000000000

# Floor division (quotient without the remainder)
print(x // y)  # Output: 0
```

Comparison operations:

```
x = 10
y = 20

# Equal to
print(x == y)  # Output: False

# Not equal to
print(x != y)  # Output: True

# Greater than
print(x > y)  # Output: False

# Less than
print(x < y)  # Output: True

# Greater than or equal to
print(x >= y)  # Output: False

# Less than or equal to
print(x <= y)  # Output: Tr
```

Logical operations:

```
x = True
y = False

# AND
print(x and y)  # Output: False

# OR
print(x or y)  # Output: True
```

```
# NOT
print(not x)  # Output: False
```

Bitwise operations:

```
x = 10  # Binary representation: 1010
y = 20  # Binary representation: 10100

# Bitwise AND
print(x & y)  # Output: 0

# Bitwise OR
print(x | y)  # Output: 30

# Bitwise XOR
print(x ^ y)  # Output: 30

# Bitwise NOT
print(~x)  # Output: -11

# Left shift
print(x << 1)  # Output: 20

# Right shift
print(y >> 1)  # Output: 10
```