Computer Science and Software Engineering Program with a focus on Python at University:

Course 1 - Learn Python Basics:

* Introduction to Python and its use cases
* Understanding data types and variables in Python
* Working with operators and expressions in Python
* Conditional statements and loops in Python
* Working with lists, tuples, and dictionaries in Python
* Functions and modules in Python
* Understanding file I/O in Python
* Basic concepts in Object-Oriented Programming (OOP) using Python
* Building basic projects using Python

Course 2 - Python Basics Challenges:

* Introduction to Python coding challenges and competitions
* Working with conditional statements and loops
* Functions and modules
* Working with collections like lists, sets, and dictionaries
* Advanced modules and libraries like NumPy, Pandas, and Matplotlib
* Working with files and directories in Python

Course 3 - Python Beyond Basics:

* Introduction to advanced concepts in Python
* Writing Pythonic code using best practices
* Advanced concepts in OOP using Python
* Exception handling in Python
* Advanced file handling using Python
* Concurrency and parallelism in Python
* Working with databases in Python
* Building advanced projects using Python

Course 4 - Python Beyond Basics Challenges:

* Advanced Pythonic code challenges
* Advanced OOP concepts
* Exception handling and debugging
* Working with concurrency and parallelism in Python

Course 5 - Python Interview Questions:

* Introduction to common Python interview questions and best practices
* Basic interview questions and coding challenges
* Intermediate level interview questions and coding challenges
* Advanced level interview questions and coding challenges
* Theory questions related to Python and computer science fundamentals
* Practice interviews with feedback and suggestions for improvement

Overall, the program covers a range of topics in computer science and software engineering with a focus on Python. It provides a solid foundation in basic and advanced concepts of Python programming and OOP, with hands-on experience working on real-world projects and coding challenges. The program also prepares students for job interviews by covering common interview questions and providing opportunities for practice interviews with feedback.

# Course 1

Lesson 1: Introduction to Python and its use cases In this lesson, we will introduce the Python programming language and its various use cases. We will cover the basics of Python syntax, data types, variables, and how to run Python code. We will also discuss the various industries and applications where Python is used.

Lesson 2: Understanding data types and variables in Python In this lesson, we will discuss the various data types in Python, such as integers, floats, strings, and booleans. We will also learn how to use variables to store and manipulate data in Python.

Lesson 3: Working with operators and expressions in Python In this lesson, we will cover the various operators in Python, such as arithmetic, comparison, logical, and bitwise operators. We will also learn how to use expressions to perform operations on data in Python.

Lesson 4: Conditional statements and loops in Python In this lesson, we will learn about conditional statements in Python, such as if/else statements and elif statements. We will also cover loops in Python, such as while loops and for loops.

Lesson 5: Working with lists, tuples, and dictionaries in Python In this lesson, we will learn about Python's built-in data structures, such as lists, tuples, and dictionaries. We will learn how to manipulate and access data stored in these data structures.

Lesson 6: Functions and modules in Python In this lesson, we will learn how to define and call functions in Python. We will also learn how to use modules to organize and reuse code in Python.

Lesson 7: Understanding file I/O in Python In this lesson, we will learn how to read and write data to files in Python. We will cover basic file I/O operations, such as opening, reading, and writing files.

Lesson 8: Basic concepts in Object-Oriented Programming (OOP) using Python In this lesson, we will introduce the basic concepts of Object-Oriented Programming using Python. We will cover classes, objects, attributes, and methods.

Lesson 9: Building basic projects using Python In this lesson, we will put our knowledge of Python to use by building basic projects, such as a calculator or a simple game.

That concludes the first course, Learn Python Basics. Next, we will move on to the second course, Python Basics Challenges.

## Lesson 1: Introduction to Python:

Python was created by Guido van Rossum in the late 1980s as a successor to the ABC programming language. It was first released in 1991 and has since become one of the most popular programming languages in the world.

Python is a high-level, interpreted programming language, which means that it is designed to be easy to read and write, and code written in Python is executed by an interpreter rather than compiled into machine code.

Let's now take a look at the syntax of Python. Unlike some other programming languages, Python uses whitespace indentation to indicate the structure of the code. For example, to define a function in Python, we use the following syntax:

def my\_function():

# code goes here

In this example, we use the def keyword to define a function called my\_function. The colon at the end of the line indicates the start of a new block of code, which is indented using four spaces.

First, let's install Python on your computer. You can download the latest version of Python from the official website (<https://www.python.org/downloads/>). Make sure to choose the correct version for your operating system.

Once you have installed Python, open a new Python file in your preferred text editor or IDE. Let's start by writing a "Hello, world!" program:

print("Hello, world!")

Save the file with a .py extension, and then run it by typing python filename.py in the terminal or command prompt.

## Lesson 1: Introduction to Python (continued)

In this lesson, we will continue our discussion of the basics of Python programming language, including variables, data types, and basic operations.

Let's start with variables. A variable is a named storage location in a program where a value can be stored and retrieved later. In Python, you can create a variable by simply assigning a value to it using the = operator. For example:

my\_variable = 42

In this example, we create a variable called my\_variable and assign the value 42 to it. Note that in Python, you don't need to declare the type of a variable. Python is a dynamically-typed language, which means that the type of a variable is inferred at runtime.

Next, let's talk about data types in Python. Python has several built-in data types, including integers, floats, strings, booleans, and more. Here are some examples:

my\_integer = 42

my\_float = 3.14

my\_string = "Hello, world!"

my\_boolean = True

In this example, we create variables of different data types. Note that strings are enclosed in double quotes or single quotes, and booleans are either True or False.

Now, let's look at some basic operations in Python. Python supports all the standard arithmetic operators, including addition, subtraction, multiplication, and division. For example:

result = 5 + 3

result = 5 - 3

result = 5 \* 3

result = 5 / 3

In this example, we perform some basic arithmetic operations and store the result in the variable result. Note that when you divide two integers in Python 3.x, the result is a float.

Python also supports string concatenation, which is the process of combining two or more strings into a single string. For example:

greeting = "Hello"

name = "John"

message = greeting + ", " + name + "!"

In this example, we create a variable called message by concatenating the strings greeting, , , name, and !.

Let's dive deeper into the topics we've covered so far in Lesson 1.

Variables:   
  
In Python, variables are dynamically-typed, meaning that the type of a variable is inferred at runtime based on the value assigned to it. This makes Python very flexible and easy to use, but it also requires careful attention to ensure that your code works as intended.

One important thing to note is that Python variables are case-sensitive, which means that my\_variable, My\_Variable, and my\_Variable are all different variables.

Data Types: In addition to the data types we've covered (integers, floats, strings, and booleans), Python has several other built-in data types, such as lists, tuples, sets, and dictionaries.

#### Lists

Lists are ordered collections of items, which can be of any data type. You can create a list by enclosing a comma-separated sequence of items in square brackets, like this:

my\_list = [1, 2, 3, "four", 5.0]

1. To create a list in Python, you can simply enclose a comma-separated sequence of items in square brackets.

For example, let's create a list of some fruits:

fruits = ["apple", "banana", "orange", "kiwi", "grape"]

1. Accessing Elements of a List You can access individual elements of a list by their index. In Python, indexing starts at 0, so the first element of the list has an index of 0, the second has an index of 1, and so on. To access an element, simply use its index in square brackets after the list name. For example, let's access the third element of our fruits list:

print(fruits[2])

Output:

orange

1. Slicing a List You can also extract a subset of elements from a list using slicing. Slicing allows you to specify a range of indices to extract from the list. To slice a list, use the following syntax: list\_name[start\_index:end\_index]. Note that the start\_index is inclusive and the end\_index is exclusive. For example, let's slice our fruits list to extract the first three elements:

print(fruits[0:3])

Output:

['apple', 'banana', 'orange']

1. Modifying Elements of a List Lists in Python are mutable, which means you can modify individual elements of a list. To modify an element, simply assign a new value to it using its index. For example, let's change the second element of our fruits list to "pineapple":

fruits[1] = "pineapple"

print(fruits)

Output:

['apple', 'pineapple', 'orange', 'kiwi', 'grape']

1. Adding Elements to a List You can add new elements to a list using the append() method. The append() method adds a single element to the end of the list. For example, let's add "mango" to our fruits list:

fruits.append("mango")

print(fruits)

Output:

['apple', 'pineapple', 'orange', 'kiwi', 'grape', 'mango']

1. Removing Elements from a List You can remove elements from a list using the remove() method. The remove() method removes the first occurrence of the specified element from the list. For example, let's remove "kiwi" from our fruits list:

fruits.remove("kiwi")

print(fruits)

Output:

['apple', 'pineapple', 'orange', 'grape', 'mango']

1. Checking if an Element is in a List You can check if a particular element is in a list using the in keyword. The in keyword returns True if the element is in the list and False otherwise. For example, let's check if "orange" is in our fruits list:

print("orange" in fruits)

Output:

True

---------------------------------------------------------------------------------------------------------------

---------------------------------------------------------------------------------------------------------------

#### Tuples

Tuples are similar to lists, but they are immutable, which means that you can't change their contents once they are created. You create a tuple by enclosing a comma-separated sequence of items in parentheses, like this:

my\_tuple = (1, 2, 3, "four", 5.0)

In this example, we create a tuple called my\_tuple that contains five items of different data types.

Accessing Elements in a Tuple

You can access individual elements in a tuple using indexing, just like with lists. The first element in a tuple has an index of 0, the second element has an index of 1, and so on. For example:

print(my\_tuple[0]) # output: 1   
print(my\_tuple[3]) # output: "four"

You can also use negative indexing to access elements from the end of the tuple. The last element in a tuple has an index of -1, the second-to-last element has an index of -2, and so on. For example:

print(my\_tuple[-1]) # output: 5.0   
print(my\_tuple[-3]) # output: 3

Tuple Slicing

You can also use slicing to access a range of elements from a tuple. Slicing works the same way as with lists. For example:

print(my\_tuple[1:4]) # output: (2, 3, "four")  
print(my\_tuple[:3]) # output: (1, 2, 3)  
print(my\_tuple[3:]) # output: ("four", 5.0)

Note that slicing a tuple returns a new tuple.

Tuple Methods

Since tuples are immutable, they have fewer methods than lists. However, there are still some useful methods you can use with tuples.

One method is the count() method, which returns the number of times a specified value appears in the tuple. For example:

print(my\_tuple.count(2)) # output: 1   
print(my\_tuple.count("four")) # output: 1   
print(my\_tuple.count(10)) # output: 0

Another method is the index() method, which returns the index of the first occurrence of a specified value in the tuple. For example:

print(my\_tuple.index(3)) # output: 2 print(my\_tuple.index("four")) # output: 3

print(my\_tuple.index(10)) # raises a ValueError, since 10 is not in the tuple

Tuples are a useful data structure for storing collections of items that should not be modified. We have seen how to create tuples, access their elements, use slicing, and use some of their methods.

---------------------------------------------------------------------------------------------------------------

---------------------------------------------------------------------------------------------------------------

Sets are unordered collections of unique items, which can be of any data type. You can create a set by enclosing a comma-separated sequence of items in curly braces, like this:

my\_set = {1, 2, 3, "four", 5.0}

Dictionaries are collections of key-value pairs, where each key is associated with a value. You can create a dictionary by enclosing a comma-separated sequence of key-value pairs in curly braces, like this:

my\_dict = {"name": "John", "age": 30, "city": "New York"}

Basic Operations: In addition to the standard arithmetic operators, Python also supports other useful operators, such as the modulo operator (%), which returns the remainder of a division operation, and the exponentiation operator (\*\*), which raises a number to a power.

For example:

result = 5 % 2 # result is 1 result = 2 \*\* 3 # result is 8

String Methods: In addition to concatenation, Python strings have many useful built-in methods for manipulating and formatting strings.   
  
For example:

message = "hello, world" message = message.upper() # message is now "HELLO, WORLD" message = message.replace(",", " ") # message is now "HELLO WORLD"

Conclusion: In this lesson, we have covered the basics of Python programming language, including variables, data types, and basic operations. We have also discussed some of the built-in data structures in Python, such as lists, tuples, sets, and dictionaries. Finally, we have looked at some useful string methods in Python. In the next lesson, we will dive deeper into data types and variables, and explore more advanced concepts, such as control flow and functions.

Lesson 2:

Control Flow:

In this lesson, we will learn about control flow in Python, which allows us to change the order in which statements are executed based on certain conditions. We will cover if-else statements, loops, and other control flow constructs.

Let's start with if-else statements. An if statement allows us to execute a block of code if a certain condition is true. Here's an example:

x = 10

if x > 5:

print("x is greater than 5")

else:

print("x is less than or equal to 5")

In this example, we first assign the value 10 to the variable x. We then check if x is greater than 5 using the if statement. If the condition is true, the first block of code (the print statement) is executed. If the condition is false, the second block of code (the print statement inside the else statement) is executed instead.

We can also use elif statements to check multiple conditions. Here's an example:

x = 10

if x > 15:

print("x is greater than 15")

elif x > 5:

print("x is greater than 5 but less than or equal to 15")

else:

print("x is less than or equal to 5")

In this example, we first check if x is greater than 15. If the condition is true, the first block of code is executed. If the condition is false, we move on to the elif statement and check if x is greater than 5. If the condition is true, the second block of code is executed. If both conditions are false, we move on to the else statement and execute the third block of code.

That's it for this lesson on control flow. In the next lesson, we will cover functions in Python.

# Course 2

Lesson 1: Introduction to Python and challenges In this lesson, we will review the basics of Python and start working on some challenges. These challenges will test our knowledge of data types, variables, operators, and expressions in Python.

Lesson 2: Flow Control challenges In this lesson, we will work on challenges that test our knowledge of conditional statements and loops in Python.

Lesson 3: Function challenges In this lesson, we will work on challenges that test our knowledge of defining and calling functions in Python. We will also work with modules and learn how to import them.

Lesson 4: Collections challenges In this lesson, we will work on challenges that test our knowledge of working with lists, tuples, and dictionaries in Python.

Lesson 5: Module challenges In this lesson, we will work on challenges that test our knowledge of modules in Python. We will learn how to create and import our own modules.

That concludes the second course, Python Basics Challenges. Next, we will move on to the third course, Python Beyond Basics.

# Course 3

Lesson 1: Introduction to Python Beyond Basics In this lesson, we will introduce the topics covered in the Python Beyond Basics course. We will discuss how to write Pythonic code, which means writing code that is clean, readable, and efficient. We will also cover Object-Oriented Programming (OOP) concepts in more depth.

Lesson 2: Writing Pythonic Code In this lesson, we will cover various techniques for writing Pythonic code, such as using list comprehensions, generators, and decorators. We will also discuss best practices for code formatting and commenting.

Lesson 3: Object-Oriented Programming (OOP) in Python In this lesson, we will dive deeper into Object-Oriented Programming (OOP) concepts in Python. We will cover inheritance, polymorphism, and encapsulation. We will also learn about abstract classes and interfaces.

Lesson 4: Exception Handling in Python In this lesson, we will learn how to handle exceptions in Python. We will cover try/except blocks, raising exceptions, and finally blocks. We will also learn how to create custom exceptions.

Lesson 5: Working with Files in Python In this lesson, we will continue our discussion of file I/O in Python. We will cover more advanced file I/O operations, such as reading and writing CSV files and working with binary files. We will also discuss file compression and decompression.

Lesson 6: Building Projects in Python In this lesson, we will put our knowledge of Python Beyond Basics to use by building more advanced projects, such as a web scraper or a data analysis tool.

That concludes the third course, Python Beyond Basics. Next, we will move on to the fourth course, Python Beyond Basics Challenges.

# Course 4

Lesson 1: Writing Pythonic Code challenges In this lesson, we will work on challenges that test our knowledge of writing Pythonic code, such as using list comprehensions, generators, and decorators. We will also cover best practices for code formatting and commenting.

Lesson 2: Object-Oriented Programming (OOP) challenges In this lesson, we will work on challenges that test our knowledge of Object-Oriented Programming (OOP) concepts in Python, such as inheritance, polymorphism, and encapsulation. We will also learn about abstract classes and interfaces.

Lesson 3: Exception Handling challenges In this lesson, we will work on challenges that test our knowledge of handling exceptions in Python, such as using try/except blocks, raising exceptions, and finally blocks. We will also learn how to create custom exceptions.

That concludes the fourth course, Python Beyond Basics Challenges. Next, we will move on to the fifth and final course, Python Interview Questions.

# Course 5

Lesson 1: Introduction to Python Interview Questions In this lesson, we will introduce the topics covered in the Python Interview Questions course. We will discuss the different levels of interview questions, such as basic, intermediate, and advanced, and we will also cover common theory questions.

Lesson 2: Basic Python Interview Questions In this lesson, we will cover basic Python interview questions, such as what is Python, what are the data types in Python, and how do you declare a variable in Python.

Lesson 3: Intermediate Python Interview Questions In this lesson, we will cover intermediate Python interview questions, such as what are decorators in Python, what is a closure, and what is the difference between a tuple and a list.

Lesson 4: Advanced Python Interview Questions In this lesson, we will cover advanced Python interview questions, such as what is the Global Interpreter Lock (GIL), what is the difference between deep and shallow copy, and how do you optimize Python code.

Lesson 5: Theory Questions In this lesson, we will cover common theory questions that may come up in a Python interview, such as what is the difference between a stack and a queue, what is the difference between DFS and BFS, and what is Big O notation.

Lesson 6: Python Coding Challenges In this lesson, we will put our knowledge to the test by working on Python coding challenges that are commonly found in interviews.

That concludes the fifth and final course, Python Interview Questions. Congratulations on completing the entire program!