GRADAMIC

B.Tech–2nd Year Session 2023–24 Odd/Even Semester

Dr. A. P. J. Abdul Kalam Technical University
Lucknow, Uttar Pradesh

PYTHON PROGRAMMING SECURITY

BCC 302 / BCC 402

app.gradamic.com

UNIT – 1st

UNIT - I

INTRODUCTION TO PYTHON

- Python
- Python variables
- Python basic Operators
- Understanding python blocks
- Python Data Types
- Declaring and using Numeric data types: int, float etc.
- Type conversion in Python
- Operator Precedence in Python
- Python Programming Cycle
- IDE and It's Importance
- Memory Management In Python
- PEP and PEP 8

PYTHON

Python is a high-level, interpreted programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991.

Here's a breakdown of its uses and applications:

Web Development: Python is widely used in web development. Frameworks like Django and Flask provide efficient tools for building web applications, APIs, and websites.

Data Science and Machine Learning: Python has become the de facto language for data science and machine learning tasks. Libraries such as NumPy, pandas, Matplotlib, and scikit-learn provide powerful tools for data analysis, visualization, and machine learning.

Scripting: Python is often used for scripting tasks due to its ease of use and readability. It's commonly used for automating repetitive tasks, system administration, and writing utility scripts.

Game Development: Python can be used for game development, both for prototyping and for creating full-fledged games. Libraries like Pygame provide tools for developing 2D games, while other libraries like Panda3D and Unity Python API enable 3D game development.

Desktop GUI Applications: Python offers several libraries and frameworks, such as Tkinter, PyQt, and wxPython, for building desktop GUI applications.

These tools enable developers to create cross-platform desktop applications with graphical user interfaces.

Scientific Computing: Python is extensively used in scientific computing for tasks such as simulation, computational physics, and engineering. Libraries like SciPy and SymPy provide tools for scientific computing and symbolic mathematics.

Network Programming: Python is commonly used for network programming tasks, including socket programming, building network servers and clients, and interacting with network devices.

Education: Python's simplicity and readability make it an excellent choice for teaching programming to beginners. Many educational institutions use Python as an introductory language for teaching programming concepts.

Embedded Systems: Python is also used in embedded systems development, particularly in projects where ease of development and rapid prototyping are prioritized. MicroPython and CircuitPython are variants of Python optimized for microcontrollers and embedded systems.

Finance and Trading: Python is widely used in the finance industry for tasks such as quantitative analysis, algorithmic trading, and financial modeling. Libraries like Pandas and NumPy are particularly popular for these applications.

Overall, Python's versatility and rich ecosystem of libraries make it suitable for a wide range of applications across various domains.

To print "Hello World!!":

```
print("Hello World!!")
```

➔ **OUTPUT: Hello World!!**

PYTHON VARIABLES

Let's talk about Python variables! In Python, a variable is like a container that holds a value. You can think of it as a label that you attach to a specific piece of data.

Here are some key points about Python variables:

1. Variable Names: A variable name can contain letters (a-z, A-Z), digits (0-9), & underscores (_). It must start with a letter or an underscore.

Python is case-sensitive, so `my_variable` and `My_Variable` are different variables.

2. Assigning Values: You assign a value to a variable using the `=` operator.

For example:

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

3. Data Types: Unlike some other programming languages, Python is dynamically typed, meaning you don't have to declare the type of a variable.

Python determines the type based on the value assigned to it. Common data types include integers, floats, strings, lists, tuples, dictionaries, etc.

4. Reassigning Variables: You can reassign a variable to a different value. The variable will now reference the new value.

For example:

```
x = 5
x = "Hello"
```

5. Variable Scope: The scope of a variable refers to the region of the code where the variable is accessible. Variables defined inside a function are local to that function, while variables defined outside functions are global.

6. Variable Naming Conventions: Although Python doesn't enforce strict rules on variable names, it's good practice to follow naming conventions for readability.

Variable names should be descriptive and meaningful, using lowercase letters with underscores for readability (e.g., `my_variable`, `user_name`).

7. Reserved Keywords: You can't use reserved keywords as variable names, as they have special meanings in Python.

Some examples of reserved keywords include `if`, `else`, `for`, `while`, `def`, `class`, etc.

Here's a simple example demonstrating some of these concepts:

```
# Assigning values to variables
x = 5
name = "John"
# Reassigning a variable
x = "Hello"
# Printing variable values
print(x)  # Output: Hello
print(name)  # Output: John
```

Python Basic Operators

1. **Arithmetic Operators:** Used for arithmetic operations like addition, subtraction, multiplication, division, etc.

```
x = 10
y = 3
print(x + y)  # Addition
print(x - y)  # Subtraction
print(x * y)  # Multiplication
print(x / y)  # Division
print(x // y) # Floor division
print(x % y)  # Modulus (remainder)
print(x ** y) # Exponentiation
```

2. **Comparison Operators:** Used to compare values and return a Boolean result (`True` or `False`).

```
x = 10
y = 5
print(x == y)  # Equal to
print(x != y)  # Not equal to
print(x > y)   # Greater than
print(x < y)   # Less than
print(x >= y)  # Greater than or equal to
print(x <= y)  # Less than or equal to
```

3. **Logical Operators:** Used to combine conditional statements.

```
x = True
y = False
print(x and y)  # Logical AND
print(x or y)   # Logical OR
print(not x)    # Logical NOT
```

4. **Assignment Operators:** Used to assign values to variables.

```
x = 5       # Assign
x += 3      # Add and assign (x = x + 3)
x -= 2      # Subtract and assign (x = x - 2)
x *= 4      # Multiply and assign (x = x * 4)
x /= 2      # Divide and assign (x = x / 2)
x %= 3      # Modulus and assign (x = x % 3)
x //= 2     # Floor divide and assign (x = x // 2)
x **= 3     # Exponentiate and assign (x = x ** 3)
```

5. **Membership Operators:** Used to test whether a value or variable is found in a sequence (lists, tuples, sets, etc.).

```
x = [1, 2, 3, 4, 5]
print(3 in x)     # True
print(6 not in x)  # True
```

6. **Identity Operators:** Used to compare the memory locations of two objects.

```
x = [1, 2, 3]
y = [1, 2, 3]
z = x
print(x is y)     # False
print(x is z)     # True
print(x is not y)   # True
```

UNDERSTANDING PYTHON BLOCKS

In Python, blocks of code are defined by indentation. This means that indentation plays a crucial role in Python syntax, unlike in many other programming languages where blocks are defined by braces or keywords.

Here's a basic explanation:

1. **Indentation:** Python uses indentation to define blocks of code. Typically, each level of indentation is four spaces. Indentation must be consistent within the same block.

2. **Code Blocks:** Blocks of code are used for control flow statements (like if-else, for loops, while loops, etc.) and function/method definitions. When you start a new block of code (after a colon `:`), you indent the following lines. The block ends when the indentation goes back to the previous level.
3. **Whitespace:** While the standard is to use four spaces for each level of indentation, you can technically use tabs or a different number of spaces, but consistency is key. Mixing tabs and spaces can lead to errors.

Here's an example to illustrate:

1.
```
a=5
If(a==5):
    print("a is equal to 5")
```

2.
```
if(a==5):
    if(b==6):
        print("a is equal to 5 and b is equal to 6")
    # Back to the outer block
    print("outer")
    # End of the block
```

3. Creating a function:
```
def my_function(parameter):
    # This is the function block
    statement1
    statement2
    return something
    # End of the function block
```

4. And for loops:
```
for item in iterable:
    # Loop block
    statement1
    statement2
    # End of loop block
```

Python Data Types

Python supports several built-in data types. Here are some of the fundamental ones:

1. Numeric Types:

- **int** = Integer type, e.g., `5`, `-3`, `1000`.
- **float** = Floating-point type, e.g., `3.14`, `-0.001`, `2.0`.

2. Sequence Types:

- **list** = Ordered collection of items, mutable (modifiable),

e.g., `[1, 2, 3]`, `['apple', 'banana', 'orange']`.

- **tuple** = Ordered collection of items, immutable (cannot be modified),

e.g., `(1, 2, 3)`, `('apple', 'banana', 'orange')`.

- **str** = String, immutable sequence of characters,

e.g., `'hello'`, `"world"`.

3. Mapping Type:

- **dict** = Collection of key-value pairs, mutable,

e.g., `{'name': 'John', 'age': 30}`.

4. Set Types:

- **set** = Unordered collection of unique items, mutable,

e.g., `{1, 2, 3}`, `{ 'apple', 'banana', 'orange'}`.

- **frozenset :** Immutable set, e.g., `frozenset({1, 2, 3})`.

5. Boolean Type:

- **bool** = Represents Boolean values `True` or `False`.

6. None Type:

- **None** = Represents the absence of a value or a null value, similar to `null` in other languages.

Python also supports complex numbers (`complex`), which consist of a real and an imaginary part.

You can check the type of a variable using the `type()` function.

For example:

```
x = 5
print(type(x))  # Output: <class 'int'>
```

DECLARING & USING DATA TYPES: INT, FLOAT, ETC.

- **Integer (int):** Integers are whole numbers without any decimal point. You can declare integers simply by assigning a value without any fractional part.

```
x = 5
y = -10
```

- **Float (float):** Floats are numbers with a decimal point or in exponential form. You can declare floats by including a decimal point or by using scientific notation with 'e' or 'E'.

```
a = 3.14
b = -0.001
c = 2.0
d = 6.022e23  # Avogadro's number (6.022 x 10^23)
```

- **Basic Arithmetic Operations:** You can perform arithmetic operations on numeric data types like `int` and `float` using operators such as **`+`, `-`, `*`, `/`, `%` (modulo), `**` (exponentiation).**

```
# Addition
sum_result = 5 + 3.14  # Result: 8.14
# Subtraction
difference = 10 – 3  # Result: 7
# Multiplication
product = 2 * 4.  # Result: 9.0
# Division
quotient = 10 / 3  # Result: 3.33333...
# Modulo
remainder = 10 % 3
    # Result: 1 (remainder of 10 divided by 3
# Exponentiation
result = 2 ** 3
    # Result: 8 (2 raised to the power of 3)
```

TYPE CONVERSION IN PYTHON

Type conversion, also known as typecasting.

It is the process of converting one data type to another. Python provides built-in functions to facilitate type conversion between different data types.

Here are some commonly used type conversion functions:

1.int(x): Converts x to an integer. If x is a float, it truncates towards zero.

```
float_num = 3.14
int_num = int(float_num)  # int_num is now 3
```

2.float(x): Converts x to a floating-point number.

```
int_value = 5
float_value = float(int_value)  # float_value is now 5.0
```

3.str(x): Converts x to a string.

```
integer_value = 10
string_value = str(integer_value)  # string_value is now '10'
```

4.bool(x): Converts x to a Boolean value. Returns False if x is False, 0, None, an empty sequence (e.g., '', [], (), {}), or an object of a numeric type with a value of zero.

Otherwise, it returns True.

```
number = 10
bool_value = bool(number)  # bool_value is now True
```

5.list(x): Converts x to a list. x should be an iterable like a string, tuple, set, etc.

```
string_value = 'hello'
list_value = list(string_value)  # list_value is now ['h', 'e', 'l', 'l', 'o']
```

6.tuple(x): Converts x to a tuple.

```
list_value = [1, 2, 3]
tuple_value = tuple(list_value)  # tuple_value is now (1, 2, 3)
```

7.set(x): Converts x to a set.

```
list_value = [1, 2, 2, 3, 3, 3]
set_value = set(list_value)  # set_value is now {1, 2, 3}
```

8.dict(x): Creates a dictionary from an iterable of key-value pairs.

```
list_of_tuples = [('a', 1), ('b', 2), ('c', 3)]
dict_value = dict(list_of_tuples)
# dict_value is now {'a': 1, 'b': 2, 'c': 3}
```

OPERATOR PRECEDENCE IN PYTHON

Operator precedence determines the order in which operators are evaluated in an expression.

Python follows a set of rules to determine the precedence of operators.

Here's a summary of the operator precedence in Python, listed from highest precedence to lowest:

1.Parentheses `( )` : Parentheses are used to override the default precedence of operators. Expressions within parentheses are evaluated first.

2.Exponentiation `**` : Exponentiation operator has the highest precedence.

3.Unary operators : Unary plus `+`, unary minus `-`, and logical negation `not`.

4.Multiplication `*`, Division `/`, Floor division `//`, and Modulus `%` : Multiplication, division, floor division, and modulus operators have the same precedence and are evaluated from left to right.

5.Addition `+` and Subtraction `-` : Addition and subtraction operators have the same precedence and are evaluated from left to right.

6.Bitwise Shifts `<<`, `>>` : Bitwise left and right shift operators have the same precedence.

7.Bitwise AND `&` : Bitwise AND operator.

8.Bitwise XOR `^` : Bitwise exclusive OR operator.

9.Bitwise OR `|` : Bitwise OR operator.

10.Comparison Operators `==`, `!=`, `>`, `<`, `>=`, `<=`, `is`, `is not`, `in`, `not in` : Comparison operators have lower precedence than bitwise operators.

11.Boolean NOT `not` : Logical NOT operator.

12.Boolean AND `and` : Logical AND operator.

13.Boolean OR `or`: Logical OR operator.

Operators with higher precedence are evaluated before operators with lower precedence.

In case of operators with the same precedence, evaluation proceeds from left to right.

Here's a simple example illustrating operator precedence:

```
result = 2 + 3 * 4
# The multiplication is performed first, then the addition.
# Result: 14
```

To ensure clarity, you can use parentheses to explicitly specify the order of evaluation:

```
result = (2 + 3) * 4
# Addition is performed first due to parentheses.
# Result: 20
```

PROGRAMMING CYCLE OF PYTHON

The "programming cycle" in Python, often referred to as the software development cycle or the coding process, typically involves several stages:

1.Problem Understanding: This is where you identify the problem you want to solve or the task you want to accomplish. You need to clearly understand the requirements and constraints.

2.Algorithm Design: Once you understand the problem, you design an algorithm to solve it. This involves breaking down the problem into smaller, manageable steps or instructions.

3.Coding: This is the stage where you write the actual Python code based on the algorithm you've designed. You use Python syntax and built-in functions to implement your solution.

4.Testing: After writing the code, you need to test it to ensure that it behaves as expected and produces the correct output for different inputs and edge cases. Testing can involve unit testing, integration testing, and system testing.

5.Debugging: If your code doesn't work as expected during testing, you need to identify and fix the errors, or bugs, in your code. This process is called debugging, and it often involves using debugging tools, print statements, and logic analysis to locate and resolve issues.

6.Optimization: Once your code is working correctly, you may want to optimize it for performance or efficiency. This could involve refactoring your code, using more efficient algorithms or data structures, or parallelizing computations.

7.Documentation: It's essential to document your code to make it understandable to others (including your future self). This includes adding comments to explain the purpose of each function or section of code, as well as writing docstrings for functions to describe their inputs, outputs, and behavior.

8.Deployment: Finally, once your code is complete, tested, and documented, you can deploy it for use in production or share it with others. Deployment might involve packaging your code into a distributable format, such as a Python package or executable, and setting up any necessary infrastructure or dependencies.

This cycle is iterative and may involve going back and forth between stages as you refine your solution, address feedback, or encounter new requirements or issues. It's essential to approach each stage deliberately and systematically to ensure the quality and effectiveness of your code.

WHAT IS AN IDE?

An **Integrated Development Environment** (IDE) is a software application that provides comprehensive facilities to computer programmers for software development. An IDE typically includes:

1.Text Editor: A text editor with features like syntax highlighting, code completion, and code formatting to make writing code easier and more efficient.

2.Compiler/Interpreter: Tools for compiling or interpreting code written in various programming languages.

3.Debugger: A debugger that allows programmers to step through code, set breakpoints, and inspect variables to find and fix errors in their code.

4.Build Automation Tools: Tools for automating the build process, such as compiling code, managing dependencies, and generating executable files or packages

5.Version Control Integration: Integration with version control systems like Git, allowing programmers to manage and track changes to their codebase.

6.Project Management Tools: Tools for managing projects, organizing files, and navigating codebases, including features like project-wide search and navigation.

WHY IDE's ARE IMPORTANT?

IDEs are important for several reasons:

1.Productivity: IDEs provide a set of tools and features that help programmers write, test, and debug code more efficiently, reducing the time and effort required to develop software.

2.Code Quality: IDEs often include features like syntax highlighting, code completion, and code analysis tools that help programmers write clean, maintainable code and identify errors and potential issues early in the development process.

3.Collaboration: IDEs often include features for version control integration, allowing programmers to collaborate on codebases more effectively by managing changes, resolving conflicts, and reviewing code.

4.Consistency: IDEs provide a consistent development environment with a unified interface and workflow, which can help improve productivity and reduce errors by providing a familiar environment for programmers to work in.

5.Learning: IDEs often include documentation, tutorials, and other educational resources that can help programmers learn new programming languages, frameworks, and tools more easily.

Overall, IDEs play a crucial role in modern software development by providing programmers with the tools and features they need to write, test, and debug code efficiently and effectively**.**

Some Python IDE's:

1. PyCharm
2. Visual Studio Code (VS Code)
3. Spyder
4. JupyterLab
5. Sublime Text
6. IDLE (Integrated Development and Learning Environment)

Some Python Debuggers:

1. pdb (Python Debugger)
2. PyCharm Debugger
3. Visual Studio Code Debugger
4. Spyder Debugger
5. Wing Debugger
6. IDLE Debugger (Integrated Development and Learning Environment)

MEMORY MANAGEMENT IN PYTHON

Memory management in Python is primarily handled by Python's memory manager, which is responsible for allocating and deallocating memory for objects as they are created and destroyed. Here are some key aspects of memory management in Python:

1.Automatic Memory Management: Python uses automatic memory management, also known as garbage collection, to handle memory allocation and deallocation. This means that programmers do not need to manually allocate and deallocate memory for objects; instead, Python's memory manager takes care of this automatically.

2.Reference Counting: Python uses reference counting as its primary memory management technique. Each object in Python has a reference count, which tracks the number of references to that object. When an object's reference count drops to zero, meaning there are no more references to it, the memory occupied by the object is deallocated.

3.Garbage Collection: In addition to reference counting, Python also employs garbage collection to reclaim memory from objects with cyclic references or objects that are part of a larger data structure. Python's garbage collector periodically runs in the background to identify and reclaim memory from unreachable objects.

4.Memory Allocation: Python's memory manager uses a variety of techniques to allocate memory for objects efficiently. For small objects, Python pre-allocates memory pools to reduce the overhead of memory allocation and deallocation. For larger objects, Python uses a more traditional memory allocation approach.

5.Memory Fragmentation: Python's memory manager can suffer from memory fragmentation, where memory becomes fragmented over time due to the allocation and deallocation of objects of different sizes. Python's memory manager includes mechanisms to mitigate memory fragmentation, such as memory pooling and periodic memory compaction.

6.Memory Profiling: Python provides tools for memory profiling, allowing programmers to analyze memory usage in their programs and identify potential memory leaks or inefficiencies. Tools like `memory_profiler` and `objgraph` can be used to profile memory usage in Python programs.

Overall, Python's memory management system is designed to provide automatic and efficient memory allocation and deallocation, allowing programmers to focus on writing code without having to worry about manual memory management.

PEP & PEP8

PEP stands for **"Python Enhancement Proposal."**

PEPs are design documents providing information to the Python community or describing a new feature for Python or its processes. The PEP process is intended to be open and collaborative, allowing Python developers to propose, discuss, and reach consensus on changes or enhancements to the Python programming language and its ecosystem.

PEP 8 is one of the most well-known and widely referenced PEPs. It is titled "Style Guide for Python Code" and provides guidelines and best practices for writing Python code to improve its readability and maintainability.

PEP 8 covers various aspects of Python coding style, including:

1.Indentation: Use 4 spaces per indentation level.

2.Tabs or Spaces: Use spaces for indentation instead of tabs.

3.Maximum Line Length: Limit lines to a maximum of 79 characters to improve readability.

4.Blank Lines: Use blank lines to separate functions, classes, and logical sections of code

5.Imports: Import modules at the top of the file, with each import on a separate line.

6.Whitespace: Use whitespace to improve readability, such as adding spaces around operators and after commas.

7.Naming Conventions: Follow naming conventions for variables, functions, classes, and modules to make code easier to understand.

8.Comments: Write clear and concise comments to explain code where necessary.

9.Docstrings: Use docstrings to provide documentation for modules, functions, classes, and methods.

10.Programming Recommendations: Follow recommendations for programming constructs, such as using list comprehensions and generator expressions where appropriate.

Following PEP 8 guidelines helps ensure consistency and readability across Python codebases, making it easier for developers to collaborate and maintain code over time.

Many Python developers and organizations adhere to PEP 8 as a standard coding style for Python projects. There are also tools and linters available that can automatically check code against PEP 8 guidelines and provide feedback on areas that need improvement.