Lecture notes on Python Programming

By Jason Pandian

Assistant Professor, Department of Information Technology

INTRODUCTION TO PYTHON PROGRAMMING

Introduction to Python Programming- Python Interpreter and Interactive Mode - Variables- Numerical types- Arithmetic operators and Expressions- Psuedo Code - Values and types: int, float, Boolean - Variables, Expressions, Statements - **Illustrative Problems.**

What is Programming?

Programming is the process of automating tasks logically and efficiently through writing code that a computer can understand and execute.

Hello, World!

In [1]:
```
print("Hello, World!")
```

```
Hello, World!
```

Print Hello World n times

In [2]:
```
n = 10
for i in range(0,n):
    print("Hello, World!")
```

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

Key Takeaway

- Programming automates tasks.
- Good programs aim for logical and efficient solutions.
- Conciseness (fewer lines of code) is a desirable aspect of effective programming.

Any Questions?

Standard Definition

Programming is the process of designing and building executable computer programs to perform specific tasks or solve particular problems. It involves creating a set of instructions that a computer can understand and execute. The primary goal of programming is to develop software that can automate tasks, process data, or provide solutions to various challenges.

Python Programming

- Python is a high-level programming interpreted language known for its simple syntax, rich libraries, and conciseness, allowing for less code.
- It is an object-oriented programming language that has gained notable popularity in web development, enterprise automation, and artificial intelligence.
- Python is also functional and procedural programming language.
- Python is also recognized as a rapid prototyping language, facilitating the transition from a conceptual idea to a functional code solution.

Pseudocode

Pseudocode is a way to represent an algorithm or a program using informal and high-level language that doesn't follow the syntax rules of a specific programming language. It's often used in the early stages of planning and

designing algorithms before implementing them in a particular language. Here's an example of pseudocode for a simple task, like finding the maximum of two numbers:

plaintext Copy code

Pseudocode for Finding the Maximum of Two Numbers

```
Function findMaximum(num1, num2):
    # Check if num1 is greater than num2
    If num1 > num2 Then
        Return num1  # num1 is the maximum
    Else
        Return num2  # num2 is the maximum
    End If
End Function
```

In this pseudocode:

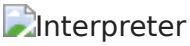
Function indicates the beginning of a function or subroutine. If is used for conditional statements. Then signifies the start of the code block executed if the condition is true. Else denotes the alternative code block executed if the condition is false. End If marks the end of the conditional statement. Remember, pseudocode is not standardized, and there are various ways to represent the same logic. Its purpose is to convey the algorithm's logic in a human-readable and understandable manner, without worrying about the syntax of a particular programming language.

Python Interpreter

A Python interpreter is a program that executes Python code. It reads and interprets Python scripts or commands, converting them into machine-readable instructions that the computer's hardware can execute. ie Interpreter is responsible for executing Python code. It serves as the bridge between the written Python script and the machine, translating the code into machine-readable instructions.

1. Interactive Mode:

The interpreter supports an interactive mode where users can enter Python commands one at a time and receive immediate feedback. This is helpful for testing small pieces of code or exploring language features.


Interpreter

In [ ]:
```
$ python
>>> print("Hello, Python!")
Hello, Python!
```

2. Script Execution

The interpreter is used to execute Python scripts stored in files. By running the interpreter followed by the script's filename, the code within the script is executed.

In [7]:
```
print("hello")
iNumber = 10
f_number = 10.5
s12 = iNumber + f_number
print(s12)
```

```
hello
20.5
```

```
$ python hello.py
```

3. Dynamic Typing

Python is dynamically typed, meaning variable types are determined at runtime. The interpreter manages this dynamic typing, allowing flexibility in variable usage.

```
>>> x = 5
>>> type(x)
<class 'int'>
```

In [6]:
```
x = 5
print(type(x))
x = "Hello"
print(type(x))
```

```
<class 'int'>
<class 'str'>
```

4.Garbage Collection

The interpreter includes a garbage collector that automatically handles memory management. It identifies and frees up memory occupied by objects that are no longer in use.

5.Standard Library

Python comes with an extensive standard library that provides ready-to-use modules and packages for various tasks. The interpreter interacts with these libraries to extend Python's functionality

In [5]:
```python
# Example Code Using Python Standard Library

# Using the `datetime` module to work with dates and times
from datetime import datetime

# Get the current date and time
current_datetime = datetime.now()

# Print the current date and time
print("Current Date and Time:", current_datetime)

# Format the date as a string
formatted_date = current_datetime.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted Date:", formatted_date)
```

```
Current Date and Time: 2024-04-03 07:47:07.355989
Formatted Date: 2024-04-03 07:47:07
```

- We import the datetime module from the standard library.
- We use datetime.now() to get the current date and time.
- We print the current date and time.
- We format the date as a string using strftime() and print the formatted date.

6.Extensibility

The interpreter is extensible, allowing the incorporation of modules written in languages like C. This feature enables developers to enhance Python's capabilities or integrate it with existing systems.

7.Platform Independence

Python interpreters are available for various platforms (Windows, macOS, Linux), making Python code portable across different operating systems.

8.Error Handling

The interpreter identifies syntax errors and runtime errors, providing feedback to the developer for debugging purposes.

In [4]:
```python
# Basic Error Handling Example

try:
    # Attempt to perform a division by zero
    result = 10 / 0
```

```
except ZeroDivisionError as e:
    # Handle the ZeroDivisionError exception
    print("Error:", e)
    print("Cannot divide by zero. Please check your input.")

except Exception as e:
    # Handle any other exceptions
    print("An unexpected error occurred:", e)

finally:
    # Code in the 'finally' block will always be executed, whether an except
    print("Execution completed.")

# The program continues to run after handling the exception
print("Program continues...")
```

```
Error: division by zero
Cannot divide by zero. Please check your input.
Execution completed.
Program continues...
```

Any Questions?

Variables

n Python, a variable is a symbolic name that refers to a value. These values can be of different types, such as numbers, strings, or more complex data structures. Here are key points about Python variables:

Variable Declaration

In Python, you don't need to explicitly declare the type of a variable. You can simply assign a value to a variable, and Python will determine its type dynamically.

In [8]:
```
s = 10
s = "Jason"
print(s)
```

```
Jason
```

Assignment Operator " = "

Variables are decleared with Assignment Operator " = ".

A simple program to add two integers. and print the result.

In [9]:
```
num1 = 10
num2 = 5
```

```
sum_result = num1 + num2
print("Result", sum_result)
```

```
Result 15
```

1.Variable Assignment

You can assign values to variables using the assignment operator (=).

In [10]:
```
# Example of variable assignment
year = 2024
name = "Tom"
```

2.Variable Naming Rules

- Variable names can contain letters, numbers, and underscores.
- They cannot start with a number.
- Variable names are case-sensitive (age and Age are different variables).

In [11]:
```
# Valid variable names
my_variable = 10
count = 5
userName = "Tom"

AGE = 25
Age = 20
age = 18

print(AGE)
print(Age)
print(age)
```

```
25
20
18
```

In [12]:
```
# Invalid variable names
2nd_place = "error"  # starts with a number
```

```
  Cell In[12], line 2
    2nd_place = "error"  # starts with a number
    ^
SyntaxError: invalid decimal literal
```

3.Dynamic Typing

Python is dynamically typed, meaning you don't need to specify the type of a variable explicitly. The interpreter infers the type based on the assigned value.

In [13]:
```
x = 10       # x is an integer
y = "hello"  # y is a string
```

4.Variable Reassignment

You can change the value of a variable by assigning it a new value.

In [14]:
```
age = 25
age = 30  # Variable 'age' is reassigned a new value
x = "python" # x is now a string
```

5.Multiple Assignment

You can assign values to multiple variables in a single line.

In [15]:
```
a, b, c = 1, 2, 3
```

6.Global and Local Variables

- Variables defined outside of functions or methods are global.
- Variables defined inside functions or methods are local.

In [16]:
```
global_variable = 10

def my_function():
    local_variable = 5
    print(global_variable)  # Accessing global variable
```

7.Variable Types

Python has various built-in data types for variables, including integers, floats, strings, lists, tuples, dictionaries, and more.

In [17]:
```
age = 25            # Integer
height = 5.9        # Float
name = "Tom"        # String
complex = 1 + 4j    # Complex Number
my_list = [1, 2, 3] # List
```

Python has No Support To Constants

In [18]:
```
YEAR = 2006
YEAR = YEAR + 1
print(YEAR)
```

```
2007
```

Numerical types

- In Python, numerical types are used to represent numeric values.
- The primary numerical types include integers (int),
- floating-point numbers (float), and
- complex numbers (complex).

1.Integer (int)

Integers represent whole numbers without any decimal points.

Example: x = 5

2.Floating-point (float)

Floating-point numbers represent numbers with decimal points or in exponential form. Example: y = 3.14

3.Complex (complex)

Complex numbers consist of a real part and an imaginary part, where the imaginary part is represented with a "j" or "J". Example: z = 2 + 3j

In [19]:
```
# Integer
integer_number = 42

# Floating-point
float_number = 3.14

# Complex
complex_number = 2 + 3j
```

These numerical types support various arithmetic operations, and you can perform calculations using them.

For example

In [20]:
```
# Arithmetic operations
result_addition = integer_number + float_number
result_multiplication = integer_number * complex_number
```

- Python also provides built-in functions to convert between different numerical types (int(), float(), complex()).
- Understanding these numerical types and their operations is fundamental for numerical computations and mathematical operations in Python.

Boolean

In Python, a boolean is a data type that represents two possible values: True or False. Booleans are fundamental in programming, especially in control flow statements and logical operations. Here are some key points about booleans in Python

1. Boolean Values

The two boolean values in Python are `True` and `False`. These values are case-sensitive, so make sure to use the correct capitalization.

In [21]:
```
x = True
y = False
```

2.Boolean Operations

Python supports various boolean operations, such as AND (and), OR (or), and NOT (not), which are used to combine or negate boolean values.

In [22]:
```
a = True
b = False
result_and = a and b  # False
result_or = a or b    # True
result_not = not a    # False
```

3.Comparison Operators

Boolean values often result from comparison operations. Common comparison operators include `==` (equal), `!=` (not equal), `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to).

In [23]:
```
num1 = 10
num2 = 5
result_equal = num1 == num2  # False
result_less_than = num1 < num2  # False
```

Arithmetic operators and Expressions in python

1.Addition (+)

Adds two values.

In [24]:
```
# Addition
add_result = 5 + 3
print(type(add_result))
print(add_result)
```

```
<class 'int'>
8
```

2.Subtraction (-)

Subtracts the right operand from the left operand.

In [25]:
```
# Subtraction
sub_result = 7 - 2
print(type(sub_result))
print(sub_result)
```

```
<class 'int'>
5
```

3.Multiplication (*)

Multiplies two values.

In [26]:
```
# Multiplication
mul_result = 4 * 6
print(type(mul_result))
print(mul_result)
```

```
<class 'int'>
24
```

4.Division (/)

Divides the left operand by the right operand. Returns a float.

In [27]:
```
# Division
div_result = 10 / 2
print(type(div_result))
print(div_result)
```

```
<class 'float'>
5.0
```

5.Floor Division (//)

Divides the left operand by the right operand and returns the largest integer less than or equal

In [28]:
```
# Floor Division
floor_div_result = 3 // 2
print(type(floor_div_result))
print(floor_div_result)
```

```
<class 'int'>
1
```

6.Modulus (%)

Returns the remainder of the division of the left operand by the right operand.

In [29]:
```
# Modulus
mod_result = 3 % 2
print(type(mod_result))
print(mod_result)
```

```
<class 'int'>
1
```

7.Exponentiation (**)

Raises the left operand to the power of the right operand.

In [30]:
```
# Exponentiation
exp_result = 2 ** 3
print(type(exp_result))
print(exp_result)
```

```
<class 'int'>
8
```

Relational Operators

- `==` (equal to)
- `!=` (not equal to)
- `>` (greater than)
- `<` (less than)
- `>=` (greater than or equal to)
- `<=` (less than or equal to)

In [31]:
```
# Relational Operators
x,y  = 8, 12
print("Equal to: ", x == y) # false
print("Not equal to: ", x != y) # true
print("Greater than: ", x > y) # false
print("Less than: ", x < y) # true
print("Greater than or equal to: ", x >= y) # false
print("Less than or equal to: ", x <= y) # true
```

```
Equal to:  False
Not equal to:  True
Greater than:  False
Less than:  True
Greater than or equal to:  False
Less than or equal to:  True
```

Logical Operators:

- `and` (logical AND)
- `or` (logical OR)
- `not` (logical NOT)

In [32]:
```
# Logical Operators
p, q = 1, 0

# 1 AND 0
print("Logical AND:", p and q)

# 1 OR 0
print("Logical OR:", p or q)

# NOT 1
print("Logical NOT:", not p)
```

```
Logical AND: 0
Logical OR: 1
Logical NOT: False
```

Increment and Decrement in Python

In Python, the ++ and -- operators do not exist. Instead, you can use the += and -= operators for incrementing and decrementing variables. Here's an example:

In [33]:
```
## Increment and Decrement in Python
x = 5

# Increment by 1
x += 1  # Equivalent to x = x + 1
print("Incremented value:", x)

# Decrement by 1
x -= 1  # Equivalent to x = x - 1
print("Decremented value:", x)
```

```
Incremented value: 6
Decremented value: 5
```

Python Statements

1. **Assignment Statements:**
   - Assign values to variables.

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

Expression Statements

Evaluate an expression and assign its result to a variable.

```
result = x + 10
```

Print Statement

Output information to the console.

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

Conditional Statements (if, elif, else)

Execute code based on a condition.

```
if x > 0:
    print("Positive")
elif x < 0:
    print("Negative")
else:
    print("Zero")
```

Loop Statements (for, while)

Execute a block of code repeatedly.

```
for i in range(5):
    print(i)

while x > 0:
    print(x)
    x -= 1
```

Pass Statement

Placeholder statement with no effect.

```
if condition:
    pass
```

Break and Continue Statements

`break` exits a loop prematurely, and `continue` skips the rest of the code in the current iteration.

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Try-Except Statements (Exception Handling)

Handle exceptions during code execution.

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Return Statement

Return a value from a function.

```
def add_numbers(a, b):
    return a + b
```

Assert Statement

Check if a condition is True.

```
assert x > 0, "x must be positive"
```

Any Questions?

These are fundamental concepts related to variables in Python. Understanding how to use and manipulate variables is crucial for effective Python programming.

Print Triangle

In [34]:
```
# Height of the triangle
height = 5
end_range = height + 1 # by adding 1 to the height. It determines the range

# Loop to print each row
for i in range(1, end_range):
    # Print spaces before the stars
    print(" " * (height - i), end="")

    # Print stars for the current row
    print("* " * i)
```

```
    *
   * *
  * * *
 * * * *
* * * * *
```

Any Questions or Doubts?

Refer the Lectures/Tutorials GitHub Page