PYTHON

```
# Why Python?
# 1. Easy to Learn and Read: # Python has a simple and elegant syntax, making it easy for beginners to grasp and write code. It emphasizes readability, reducing the cost of program maintenance.
# 2. Versatile and Powerful: # Python can be used for a wide range of applications, including web development, data analysis, machine learning, artificial intelligence, scripting, automation, and more.
# 3. Large Community and Libraries: # Python has a massive and active community of developers. This means a vast collection of libraries and frameworks are available, enabling you to accomplish tasks quickly and efficiently.
# 4. High Demand in the Job Market: # Python is one of the most popular programming languages, and it's in high demand across various industries. Learning Python can open up numerous career opportunities.
# 5. Cross-Platform Support: # Python is a cross-platform language, meaning code written on one platform can run on multiple operating systems without modification.
# 6. Ideal for Prototyping and Rapid Development: # Python's simplicity and extensive libraries make it perfect for rapid prototyping and development. It allows you to quickly test ideas and build applications faster.
# 7. Great for Data Science and AI: # Python's data manipulation and analysis libraries (such as Pandas, NumPy, and SciPy) make it an excellent choice for data science and AI projects, enabling researchers to gain insights from large datasets. # Overall, Python is a versatile, beginner-friendly, and powerful language that's used extensively in various domains, making it a valuable skill for any programmer to learn.
```

- **NUMERIC TYPES:**
  - int: Integer type (e.g., 5, -10, 100)
  - float: Floating-point type (e.g., 3.14, -2.5, 1e-5)
  - complex: Complex numbers (e.g., 2 + 3j)
- **STRING:** A sequence of characters enclosed in single ('') or double ("") quotes. (e.g., "Hello", 'Python')
- **BOOLEAN:** Represents a binary value, either True or False. Useful in conditional statements.
- **LIST:** An ordered collection that allows duplicates and is mutable. It is denoted by square brackets []. (e.g., [1, 2, 3], ['apple', 'banana'])
- **TUPLE:** Similar to lists, but immutable (cannot be changed after creation). Denoted by parentheses (). (e.g., (1, 2, 3), ('red', 'blue'))
- **SET:** An unordered collection that doesn't allow duplicates. Denoted by curly braces {}. (e.g., {1, 2, 3}, {'cat', 'dog'})
- **DICTIONARY:** A collection of key-value pairs enclosed in curly braces {}. (e.g., {'name': 'John', 'age': 25})

FIRST PROGRAM

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

VARIABLE DECLARATION

We can create a variable by assigning a value to it using the equals (=) sign

COMMENTS

- Single-line comments: #

```
# This is a single-line comment in Python
```

- Multi-line comments: ’’’WORLD
  HELLO’’’

```
"""
This is a multi-line comment (docstring) in Python.
It can span multiple lines and is often used to describe
"""
```

ASSIGNING A VARIABLE

```
# Integer variable
age = 25

# String variable
name = "John Doe"

# Float variable
salary = 50000.50

# Boolean variable
is_student = True
```

IDENTIFYING THE TYPE OF A VARIABLE

The **‘Type()’** function in python is a built-in function that allows you to determine the data type of a given object or variable. It returns a type object, representing the specific data type of the object.

```
Age = 25
Print(type(Age)) #output : <class ‘int’>

Name = ‘John Doe’
Print(type(Name)) #output : <class ‘str’>

Is_student = True
Print(type(Is_student )) #output : <class ‘bool’>
```

PYTHON

RULES FOR DEFINING A VARIABLE NAME

- Variable names can contain letters (both uppercase and lowercase), digits, and underscores.The first character of a variable name cannot be a digit. It must be a letter (a-z, A-Z) or an underscore (_).
- Python is case-sensitive so **‘myVariable’, ‘myvariable’, ‘MYVARIABLE’**, are considered three different variables.
- You cannot use python’s reserved keywords(e.g., if, else, for, while, def, class, etc.) as variable names since they have special meanings in the language.
- Using underscores in variable names is common and encouraged for readability.
- Variables with names like my_variable or user_name are examples of using underscores.
- Variable names cannot contain spaces. If you need to represent multiple words, use underscores or CamelCase notation (e.g., my_variable, user_name, firstName, lastName).
- Avoid using special characters like @, $, %, etc., in variable names. Stick to letters, digits, and underscores.
- Choose variable names that are descriptive and meaningful. This improves code readability and makes your code easier to understand.

ARITHMETIC OPERATORS

OPERATOR	DESCRIPTION	EXAMPLE	RESULT
+	ADDITION	5 + 3	8
-	SUBTRACTION	7 - 2	5
*	MULTIPLICATION	4 * 6	24
/	DIVISION	10 / 2	5.0
//	FLOOR DIVISION (INTEGER DIVISION)	10 // 3	3
%	MODULUS (REMAINDER)	10 % 3	1
**	EXPONENTIATION (POWER)	2 ** 3	8

ASSIGNED OPERATORS

OPERATOR	DESCRIPTION	EXAMPLE	EQUIVALENT TO
=	ASSIGNMENT	x = 10	x = 10
+=	ADD AND ASSIGN	x += 5	x = x + 5
-=	SUBTRACT AND ASSIGN	x -= 3	x = x - 3
*=	MULTIPLY AND ASSIGN	x *= 2	x = x * 2
/=	DIVIDE AND ASSIGN	x /= 4	x = x / 4
//=	FLOOR DIVIDE AND ASSIGN	x //= 3	x = x // 3
%=	MODULO AND ASSIGN	x %= 7	x = x % 7
**=	EXPONENTIATION AND ASSIGN	x **= 2	x = x ** 2
&=	BITWISE AND AND ASSIGN	x &= 3	x = x & 3
|=	BITWISE OR AND ASSIGN	x |= 5	x = x | 5
^=	BITWISE XOR AND ASSIGN	x ^= 6	x = x ^ 6
<<=	LEFT SHIFT AND ASSIGN	x <<= 2	x = x << 2
>>=	RIGHT SHIFT AND ASSIGN	x >>= 3	x = x >> 3

PYTHON

COMPARISION OPERATORS

OPERATOR	DESCRIPTION	EXAMPLE	RESULT
==	EQUAL	5 == 5	TRUE
!=	NOT EQUAL	5 != 3	TRUE
>	GREATER THAN	10 > 5	TRUE
<	LESS THAN	3 < 7	TRUE
>=	GREATER THAN OR EQUAL TO	8 >= 8	TRUE
<=	LESS THAN OR EQUAL TO	4 <= 2	FALSE

INPUT() FUNCTION

In python, the **'input()'** function is used to take user input from the console or command line. It allows the program to pause and wait for the user to enter some data, which is then returned as a string.

```
user_input = input("Enter your input: ")
```

-the **'input()'** function always returns a string
-by using appropriate typecasting functions like 'int()', 'float()' the input can be converted to specific data types

LOGICAL OPERATORS

OPERATOR	DESCRIPTION	EXAMPLE
AND	Logical **AND**. Returns True if both operands are True, otherwise False.	True and False returns False
OR	Logical **OR**. Returns True if at least one operand is True, otherwise False.	True or False returns True
NOT	Logical **NOT**. Returns True if the operand is False, and vice versa.	not True returns False

LEN FUNCTION

```
my_string = "Hello, World!"
length = len(my_string)
print(length) # Output: 13
```

STRINGS

```
#string creation
```

Strings in python are sequences of characters and are one of the fundamental data types in the language. They are used to represent text and are enclosed in either single quotes('') or double quotes("").

```
# Single-quoted string
single_quoted_string = 'This is a single-quoted string.'

# Double-quoted string
double_quoted_string = "This is a double-quoted string."

# Triple-quoted string (for multi-line strings)
multi_line_string = '''This is a multi-line string.
It can span multiple lines without using escape characters.
'''
```

PYTHON

```python
# Escaping single and double quotes
escaped_string = 'He said, "Don\'t do that.“’

print(escaped_string)

# output:
He said, "Don't do that."

# Using triple quotes to include both single and double quotes without escaping
mixed_quotes = """She said, "It's okay."""

print(mixed_quotes)

# output:
She said, "It's okay."

# Raw string (ignores escape characters)
```

A raw string is a string that ignores escape charaters. It is created by prefixing the string with ‘r’ or ‘R’.

```python
raw_string = r'C:\Users\John\Documents’
print(raw_string)

# output:
C:\Users\John\Documents

# Concatenating strings
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name

print(full_name)


# output:
John Doe
```

```python
# String formatting using f-strings
```

-In this code, an f-string(formatted string literal) to format the string is used. The variables ‘first name’,’last_name’, and ‘age’ are embedded with curly braces ‘{}’ in the string.
-The f-string evaluates these expressions and replaces them with their corresponding values.

```python
age = 30
formatted_string = f"My name is {first_name} {last_name} and I am {age} years old."
print(formatted_string)

# string formatting using the format() method in Python.
name = "Alice"
age = 30
# Using format()
formatted_string = "My name is {}, and I am {} years old.".format(name, age)
#"My name is Alice, and I am 30 years old."

# output:
My name is John Doe and I am 30 years old.

# String indexing and slicing

- Indexing allows  to access individual characters in a string by their position in the sequence.
- string indexing starts from 0 for the first character and counts up incrementally.
- Negative indices count from the end of the string, where -1 represents the last character, -2 represents the second-to-last character, and so on.

my_string = "Hello, World!"

print(my_string[0]) # Output: 'H' (First character)
print(my_string[4]) # Output: 'o' (Fifth character)
print(my_string[-1]) # Output: '!' (Last character)
print(my_string[-3]) # Output: 'r' (Third-to-last character)
```

PYTHON

Slicing allows you to extract a portion of the original string. It is done by specifying a range of indices using the syntax **[start:end]**. The result includes characters from the **start** index (inclusive) to the **end** index (exclusive).

```
my_string = "Hello, World!"

print(my_string[0:5]) # Output: 'Hello' (Characters from index 0 to 4)
print(my_string[7:]) # Output: 'World!' (Characters from index 7 to the end)
print(my_string[:5]) # Output: 'Hello' (Characters from the beginning to index 4)
print(my_string[-6:]) # Output: 'World!' (Last 6 characters)
```

- slicing returns a new string and does not modify the original string.
- If the start index is not specified, it defaults to 0, and if the end index is not specified, it defaults to the end of the string.
- Additionally, slicing allows for the use of a step value, but when omitted, the step defaults to 1.

```
my_string = "Hello, World!"

print(my_string[::2]) # Output: 'Hlo ol!' (Characters with a step of 2)
print(my_string[::-1]) # Output: '!dlroW ,olleH' (Reverse the string)

# String methods in Python
# 1. capitalize(): Converts the first character of the string to uppercase.
string1 = "hello, world!"
result1 = string1.capitalize()
print(result1) # Output: "Hello, world!"

# 2. upper(): Converts all characters in the string to uppercase.
string2 = "hello, world!"
result2 = string2.upper()
print(result2) # Output: "HELLO, WORLD!"

# 3. lower(): Converts all characters in the string to lowercase.
string3 = "Hello, World!"
result3 = string3.lower()
print(result3) # Output: "hello, world!"

# 4. title(): Converts the first character of each word to uppercase.
string4 = "hello, world!"
result4 = string4.title()
print(result4) # Output: "Hello, World!"

# 5. swapcase(): Swaps the case of all characters in the string.
string5 = "Hello, World!"
result5 = string5.swapcase()
print(result5) # Output: "hELLO, wORLD!"

# 6. strip(): Removes leading and trailing whitespace characters from the string.
string6 = " Hello, World! "
result6 = string6.strip()
print(result6) # Output: "Hello, World!"

# 7. replace(): Replaces occurrences of a substring with another substring.
string7 = "Hello, World!"
result7 = string7.replace("Hello", "Hi")
print(result7) # Output: "Hi, World!"

# 8. split(): Splits the string into a list of substrings using a specified separator.
string8 = "apple,banana,orange"
result8 = string8.split(",")
print(result8) # Output: ['apple', 'banana', 'orange’]

# 9. join(): Joins elements of a list into a single string using a specified separator
 list9 = ["apple", "banana", "orange"]
result9 = ",".join(list9)
print(result9) # Output: "apple,banana,orange"

# 10. isalpha(): Returns True if all characters in the string are alphabetic.
string10 = "Hello"
result10 = string10.isalpha()
print(result10) # Output: True
```

PYTHON

```python
# 11. isdigit(): Returns True if all characters in the string are digits.
string11 = "123"
result11 = string11.isdigit()
print(result11) # Output: True

# 12. startswith(): Returns True if the string starts with a specified substring.
string12 = "Hello, World!"
result12 = string12.startswith("Hello")
print(result12) # Output: True

# 13. endswith(): Returns True if the string ends with a specified substring.
string13 = "Hello, World!"
result13 = string13.endswith("World!")
print(result13) # Output: True

# 14. count(): returns the number of non-overlapping occurrences of substring
within a string.
my_string = "Hello, Hello, Hello, World!"
substring = "Hello"

count_result = my_string.count(substring)
print(count_result)  #output: 3

# 15. find(): returns the lowest index of the first occurrence of a substring
within the string. If the substring is not found, it returns -1.

my_string = "Hello, World!"
substring = "World"

find_result = my_string.find(substring)
print(find_result) # Output: 7

my_string = "Hello, World!"
substring = "Universe"

find_result = my_string.find(substring)
print(find_result) # Output: -1
```

```python
# Escape Sequences in Python

# Newline
print("Hello\nWorld!")
# Output:
# Hello
# World!

# Tab
print("Name:\tAlice")
# Output: Name:           Alice

# Backslash
print("This is a backslash: \\")
# Output: This is a backslash: \

# Single Quote
print('I\'m learning Python.’)
# Output: I'm learning Python.

# Double Quote
print("She said, \"Hello!\"")
# Output: She said, "Hello!"

# Carriage Return
print("Hello\rWorld!")
# Output: World!ollo

# Vertical Tab
print("Hello\vWorld!")
# Output: Hello
World!

# Backspace
print("Hello\bWorld!")
# Output: HellWorld!
```

```python
# Alert/Bell
print("Hello\aWorld!")
# Output: Hello World! (Depending
on the system, this may create an
audible alert sound)

# Unicode Escape (using \u)
print("\u03A9") # Greek capital
letter Omega
# Output: Ω

# Unicode Escape (using \U)
print("\U0001F604") # Smiling Face
with Open Mouth and Smiling Eyes
emoji
# Output: 😄
```