Assignment No-6

Q.1. What are keywords in python? Using the keyword library, print all the python keywords.

Solution: Keywords in Python are reserved words that have special meanings and cannot be used as identifiers (e.g., variable or function names). They are used to define the syntax and structure of the language.

In Python, you can access the list of all keywords using the `keyword` library. Here's how you can print all the Python keywords:

```
import keyword
keywords = keyword.kwlist
print(keywords)
```

The `keyword.kwlist` returns a list of all the Python keywords, and `print(keywords)` will print the list for us.

Q.2. What are the rules to create variables in python?

In Python, you can create variables using the following rules:

a) Variable names must start with a letter (a-z, A-Z) or an underscore (_).

b) The rest of the variable name can consist of letters, numbers (0-9), or underscores (_)

but cannot contain spaces or special characters.

c) Variable names are case-sensitive, meaning "name" and "Name" are considered two different variables.

d) You cannot use reserved keywords as variable names. For example, "if", "while", "for" are reserved keywords and cannot be used as variable names.

e) Variable names should be descriptive and meaningful, helping to understand the purpose of the variable.

f) Variable names should follow the PEP 8 style guide, which suggests using lowercase letters and underscores to separate words in variable names, for example, "my_variable" instead of "myVariable".

Here are some examples of valid variable names:

- my_variable
- name
- _score
- num2
- x

And here are some examples of invalid variable names:

- 1variable (starts with a number)

- my variable (contains a space)

- my-variable (contains a hyphen)

- if (using a reserved keyword)

Q.3. What are the standards and conventions followed for the nomenclature of variables in Python to improve code readability and maintainability?

Solution: In Python, several standards and conventions are commonly followed for variable naming to enhance code readability and maintainability. The fundamental guidelines are as follows:

a) Use descriptive names: Choose variable names that accurately reflect their purpose or content. This helps in understanding the code and reduces the need for excessive comments.
b) Use lowercase with words separated by underscores: Variable names should be in lowercase letters; if needed, multiple words should be separated by underscores. For example: `count`, `first_name`.
c) Avoid using single-character names: Except for simple loop variables (e.g., `i`, `j`), it is generally best to use meaningful names. This improves the code's readability and makes it easier to understand its purpose.
d) Avoid using reserved keywords: Python has a set of reserved keywords with specific language meanings. Avoid using these keywords as variable names to prevent conflicts and confusion.
e) Use uppercase for constants: If a variable should be treated as a constant (a value that should not be changed), use uppercase letters with underscores to make it stand out. For example: `MAX_DEPTH`, `PI`.
f) Be consistent: Maintain consistency in naming conventions throughout your codebase, including function names, variable names, and class names. This makes your code more readable and understandable.
g) Follow the naming convention for specific cases: There are a few additional conventions for specific variables. For example, class names should follow the

CapWords convention (capitalize the first letter of each word without underscores), module-level constants should be defined in uppercase, etc.

h) Use meaningful abbreviations: If you need to use abbreviations in your variable names, ensure they are meaningful and widely understood within the relevant domain. Avoid obscure or ambiguous abbreviations.

Overall, the goal is to choose clear, concise, and meaningful names, making it easier for others (and yourself) to understand and maintain the code in the long run.

Q.4. What will happen if a keyword is used as a variable name?

Solution: If a keyword is used as a variable name in a programming language, it will generally result in a syntax error. This is because keywords are reserved for specific purposes and have special meanings in the language. The compiler or interpreter will not recognize the keyword used as a variable and will flag it as an error.

For example, "if" is a keyword used for conditional statements in Python. If you try to use it as a variable name, you will get a syntax error:

```
if = 10  # SyntaxError: invalid syntax
```

Similarly, in languages like Java or C++, keywords like "int", "class", or "return" have special meanings and cannot be used as variable names.

To avoid such errors, choosing variable names that are not keywords in the programming language being used is essential.

Q.5. For what purpose def keyword is used?

Solution: The "def" keyword is used to define a function in Python. It is followed by the name of the function, the parameters it takes (if any), and a colon (:). The following indented block of code is the function's body, where the function's actual logic is written.

Q.6. What is the operation of this special character '\'?

Solution: The special character '\' is known as the backslash. It is primarily used as an escape character in many programming languages and systems. It indicates that the character following it has a special meaning and should be interpreted differently.

Some common uses of the backslash are:

a) Escape sequences: It is used to create special characters that cannot be directly represented in a string. For example, '\n' represents a newline character, '\t' represents a tab character, '\r' represents a carriage return, etc.
b) File paths: In file systems, the backslash denotes directory hierarchies. For example, 'C:\Program Files\' is a standard file path in Windows systems.
c) Regular expressions: In regular expressions, the backslash is used to give special meaning to certain characters. For example, '\d' represents any digit, '\w' represents any alphanumeric character, etc.
d) Escape characters: Sometimes, it is used to escape special characters that have a specific meaning. For example, if you want to use a double quote (") within a string surrounded by double quotes, you would escape it as '\"'.

It is important to note that the usage and behavior of the backslash can vary depending on the programming language or system being used.

Q.7. Give an example of the following conditions:
(i) Homogeneous list
(ii) Heterogeneous set
(iii) Homogeneous tuple
Solution: (i) Homogeneous list: [1, 2, 3, 4, 5]
A homogeneous list is a list that contains elements of the same type. In this example, all the elements are integers.
(ii) Heterogeneous set: {1, "hello", True, 3.14}
A heterogeneous set is a set that contains elements of different types. In this example, the set contains an integer, a string, a boolean, and a float.
(iii) Homogeneous tuple: (10, 20, 30, 40, 50)
A homogeneous tuple is a tuple that contains elements of the same type. In this example, all the elements are integers.

Q.8. Explain the mutable and immutable data types with proper explanation & examples.

Solution: Mutable data types are data types that can be modified after they are created. This means that the value of these types can be changed without creating a new object. On the other hand, immutable data types cannot be modified once they are created. If you try to change the value of an immutable data type, a new object with the modified value is created.

Examples of mutable data types in Python include lists and dictionaries. Let's consider the following code snippet:

```
my_list = [1, 2, 3]

print(my_list)  # Output: [1, 2, 3]

my_list.append(4)

print(my_list)  # Output: [1, 2, 3, 4]
```

In this example, we create a list `my_list` with the values [1, 2, 3]. Since lists are mutable, we can modify it by calling the `append()` method to add the value 4 to the end of the list. The original list is modified and we get the output [1, 2, 3, 4]. This demonstrates how mutable data types can be changed after they are created.

Now let's consider an example of an immutable data type, which includes numbers, strings, and tuples in Python.

```
my_string = "Hello, World!"

print(my_string)  # Output: Hello, World!

my_string_modified = my_string.upper()

print(my_string_modified)  # Output: HELLO, WORLD!

print(my_string)  # Output: Hello, World!
```

In this example, we create a string `my_string` with the "Hello, World!". String objects are immutable, so when we call the `upper()` method to convert the string to uppercase, a new object is created with the modified value. The original string remains unchanged, and we get the output "HELLO, WORLD!" for `my_string_modified` while `my_string` still retains its original value "Hello, World!".

So, to summarize, mutable data types can be modified without the need to create a new object, while immutable data types cannot be modified, and any modification requires creating a new object.

Q.9. Write a code to create the given structure using only a for loop.

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

Solution: Here is a Python code that uses a for loop to create the given structure:

```
num_rows = 5

for i in range(num_rows):
    for j in range(num_rows-i-1):       # for adding spaces before each "*"
        print(" ", end="")
    for j in range(2*i+1):              # for adding "*" based on the row number
        print("*", end="")
    print()                             # for printing a new line after each row
```

Output:

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

Explanation:

- The outer for loop iterates over each row starting from 0 up to `num_rows-1`.
- The inner for loop is responsible for adding spaces before each "*" in each row. The number of spaces decreases with each row. It runs `num_rows-i-1` times.
- The second inner loop adds the "*" characters. The number of "*" characters increases with each row and is equal to `2*i + 1`.
- After each row printing, a new line is printed using the `print()` function.

Q.10. Write a code to create the given structure using a while loop.

```
|||||||||
 |||||||
  |||||
   |||
    |
```

Solution: Here is a code snippet that uses a while loop to create the given structure:

```
num_pipes = 9
num_spaces = 0

while num_pipes >= 1:
   print(" " * num_spaces + "|" * num_pipes)
   num_pipes -= 2
   num_spaces += 1
```

This code initializes `num_pipes` to 9 and `num_spaces` to 0. Then, it enters a while loop that continues as long as `num_pipes` is greater than or equal to 1.

In each iteration of the loop, the code prints a number of spaces equal to `num_spaces` followed by a number of pipes equal to `num_pipes`. Then, it decreases `num_pipes` by 2 and increases `num_spaces` by 1.
The result of running this code will be the desired structure:

```
|||||||||
 |||||||
  ||||||
   ||||
    ||
```