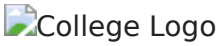

Lecture notes on Python Programming

By Jason Pandian
Assistant Professor, Department of Information Technology

DATA TYPES, CONTROL FLOW, STRINGS

Control Flow -conditional (if), Alternative (if-else), Chained conditional (if-elif-else)- Iteration: state, while, for, break, continue, pass - Strings: string slices, immutability, string functions and methods, string module, Regular expression, Pattern matching . - Illupdated_strative Problems.

What is Control Flow?

Control flow in Python refers to the order in which statements and instructions are executed in a program. It determines the flow of execution based on certain conditions or loops.

If Statement

Used for executing a block of code if a specified condition is True.

In [ ]:
```
x = 10
if x > 0:
    print("x is positive")
```

If-Else Statement

Executes one block of code if the condition is True, and another block if it's False.

In [ ]:
```
y = -5
if y > 0:
    print("y is positive")
else:
    print("y is non-positive")
```

Chained Conditional (If-Elif-Else)

Allows checking multiple conditions in sequence.

In [ ]:
```
z = 0
if z > 0:
    print("z is positive")
elif z < 0:
    print("z is negative")
else:
    print("z is zero")
```

Iteration

Iteration in programming is the repetition of a set of instructions or code block until a certain condition is met.

State

- "state" - **status** refers to the current values of variables and attributes within an object or program, representing its configuration at a given moment.
- It encapsulates data such as variable values, object attributes, and program execution status, crucial for understanding and managing program behavior and interactions.

While Loop

Repeats a block of code as long as a given condition is True.

In [ ]:
```
count = 0
while count < 5:
    print("Count:", count)
    count += 1
```

In [ ]:
```
count = 0
while count <= 5:
    print("Count:", count)
    count += 1
```

In [ ]:
```
count = 5
while count >= 0:
    print("Count:", count)
    count -= 1
```

In [ ]:
```
while 1 :
    print("True")
```

In [ ]:
```
count = 5
while count >= 0:

```

```
    print("Count:", count)
    count += 1 # error count = count + 1
```

For Loop

Iterates over a sequence (e.g., list, tuple, string) or other iterable objects.

In [ ]:
```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

In [ ]:
```
for i in range(1,11):
    print(i)
```

Break Statement

Terminates the loop prematurely when a certain condition is met.

In [ ]:
```
for num in range(10):
    if num == 5:
        break
    print(num)
```

Continue Statement

Skips the rest of the code inside the loop for the current iteration and moves to the next one.

In [ ]:
```
for num in range(5):
    if num == 2:
        continue
    print(num)
```

Pass Statement

Acts as a placeholder; it does nothing and is used when a statement is syntactically required.

In [ ]:
```
for item in fruits:
    # Some code here
    pass  # Placeholder, does nothing
```

Strings

In Python, strings are **sequences of characters, immutable**, and can be manipulated using various built-in **functions and methods**.

String Traversal

In [ ]:
```
for c in "Hello, World":
    print(c)
```

String Slicing

In [ ]:
```
my_str = "Python Lang"
print(my_str[0])    # Output: 'P'
```

In [ ]:
```
print(my_str[0:1])    # Output: 'P'
```

In [ ]:
```
print(my_str[0:5])    # Output: 'Pytho'
```

In [ ]:
```
print(my_str[0:6])    # Output: 'Python'
```

In [ ]:
```
print(my_str[6:7])    # Output: ' '
```

In [ ]:
```
print(my_str[7:8])    # Output: 'L'
```

In [ ]:
```
print(my_str[-1:])    # Output: 'g'
```

In [ ]:
```
print(my_str[:])      # Output: 'Python Lang' (Full string)
```

In [ ]:
```
print(my_str[:6])     # Output: 'Python' (From the beginning up to index 6)
```

In [ ]:
```
print(my_str[7:])     # Output: 'Lang' (From index 7 to the end)
```

In [ ]:
```
print(my_str[-4:])    # Output: 'Lang' (Last 4 characters)
```

In [ ]:
```
print(my_str[::2])    # Output: 'Pto ag' (Every other character) ->stepsize
```

In [ ]:
```
print(my_str[::-1])   # Output: 'gnaL nohtyP' (Reverse the string)
```

In [ ]:
```
# Example of strings as sequences of characters
my_string = "Hello, World!"
print("Characters in the string:", my_string)  # Output: Hello, World!

# Accessing individual characters in the string
print("First character:", my_string[0])  # Output: H
print("Last character:", my_string[-1])  # Output: !
```

In [ ]:
```
# Example of string immutability
# Attempting to modify a character in the string will result in an error
my_string[0] = 'h'  # Raises TypeError: 'str' object does not support item a
```

In [ ]:
```
# String manipulation example
# Replacing a substring
modified_string = my_string.replace("Hello", "Hi")
```

```
print("Modified string:", modified_string)  # Output: Hi, World!
type(modified_string)
```

In [ ]:
```
# Splitting the string into a list of substrings
split_string = my_string.split(',')
print("Split string:", split_string)  # Output: ['Hello', ' World!']
type(split_string)
```

In [ ]:
```
# Joining the substrings back into a single string
joined_string = ','.join(split_string)
print("Joined string:", joined_string)  # Output: Hello, World!
type(joined_string)
```

Key Takeaway

- Sequence of characters.
- Immutable.
- But can be manupulated

Under the hood for Simple Replace

In [ ]:
```
my_string = "Hello, World!"
print(len(my_string))
```

In [ ]:
```python
def strReplace(my_string, given_str, updated_str):
    # Initialize an empty string to store the modified string
    new_string = ""

    # Initialize an index variable to iterate through the original string
    i = 0

    print("length of th given_str: ",len(given_str))

    # Iterate through the original string
    while i < len(my_string):
        # Check if the current substring matches the substring to be replace
        if my_string[i:i + len(given_str)] == given_str:
            # If yes, append the replacement substring to the new string
            new_string += updated_str
            # Update the index to skip the replaced substring
            i += len(given_str)
        else:
            # If no match, append the current character to the new string
            new_string += my_string[i]
            # Move to the next character in the original string
            i += 1

    # Return the modified string
    return new_string

# Test the function
print(strReplace("Hello, World!", "Hello", "hi"))
```

In [ ]:
```python
"hello" + " " + "world"
```

String functions and methods

In Python, string functions and methods are tools used to manipulate strings, which are sequences of characters.

String Functions

Definition: String functions are standalone functions that operate on strings and return a result. They do not modify the original string. Example: len() is a string function that returns the length of a string.

len()

- `len()` : Returns the length of a string.

In [6]:
```python
string = "Hello"
print(len(string))  # Function Example: 5
```

String Methods

Definition: String methods are functions that are called on string objects and operate directly on them. They can modify the original string and/or return a modified version of it. Example: .lower() is a string method that converts all characters in a string to lowercase.

1. lower()

`lower()` : Converts all characters in a string to lowercase.

In [7]:
```
string = "HELLO"
print(string.lower())  # Method Example: HELLO
```

```
hello
```

2. upper()

`upper()` : Converts all characters in a string to uppercase.

In [8]:
```
string = "hello"
print(string.upper())  # Method Example: hello
```

```
HELLO
```

3. strip()

`strip()` : Removes leading and trailing whitespace from a string.

In [9]:
```
string = "   hello   "
print(string.strip())  # Method Example: hello
```

```
hello
```

4. split()

`split()` : Splits a string into a list of substrings based on a delimiter.

In [13]:
```
string = "apple,banana,orange"
fruits = string.split(',')
print(fruits)  # Method Example: ['apple', 'banana', 'orange']
```

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

5. join()

`join()` : Joins elements of an iterable into a string, using the string as a separator.

In [15]:
```
fruits = ['apple', 'banana', 'orange']
string = ','.join(fruits)
```

```
print(string)  # Method Example: apple,banana,orange
print(type(string))
```

```
apple,banana,orange
<class 'str'>
```

6. replace()

`replace()` : Replaces occurrences of a substring with another substring.

In [16]:
```
string = "Hello, World!"
new_string = string.replace("World", "Universe")
print(new_string)  # Method Example: Hello, Universe!
```

```
Hello, Universe!
```

7. startswith()

`startswith()` : Checks if a string starts with a specified prefix.

In [18]:
```
string = "Hello, World!"
print(string.startswith("Hello"))  # Method Example: True
```

```
False
```

8. endswith()

`endswith()` : Checks if a string ends with a specified suffix.

In [20]:
```
string = "Hello, World!"
print(string.endswith("World!"))  # Method Example: True
```

```
True
```

9. find()

`find()` : Searches for a substring within a string and returns the lowest index where it's found.

In [21]:
```
string = "Hello, World!"
print(string.find("World"))  # Method Example: 7
```

```
7
```

10.count()

`count()` : Counts the number of occurrences of a substring within a string.

In [22]:
```
string = "apple, banana, apple, orange"
print(string.count("apple"))  # Method Example: 2
```

```
2
```

11. capitalize()

`capitalize()` : Converts the first character of a string to uppercase and the rest to lowercase.

In [25]:
```
string = "hello world"
print(string.capitalize())  # Method Example: Hello world
print(string.title())
```

```
Hello world
Hello World
```

12. casefold()

`casefold()` : Converts a string to lowercase, suitable for case-insensitive comparisons.

In [26]:
```
string = "Hello World"
print(string.casefold())  # Method Example: hello world
```

```
hello world
```

13. isdigit()

`isdigit()` : Checks if all characters in a string are digits.

In [33]:
```
string = "123"
print(string.isdigit())  # Method Example: True
```

```
True
```

14. islower()

`islower()` : Checks if all characters in a string are lowercase.

In [34]:
```
string = "hellO"
print(string.islower())  # Method Example: True
```

```
False
```

15. isupper()

`isupper()` : Checks if all characters in a string are uppercase.

In [35]:
```
string = "HELLO"
print(string.isupper())  # Method Example: True
```

```
True
```

16. isspace()

`isspace()` : Checks if all characters in a string are whitespace.

In [38]:
```
string = "    "
print(string.isspace())  # Method Example: True
```

```
True
```

17. istitle()

istitle(): Checks if a string is titlecased (i.e., every word starts with an uppercase character and the rest are lowercase).

In [39]:
```
string = "Hello World"
print(string.istitle())  # Method Example: True
```

```
True
```

18. split()

`split()` : Splits a string into a list of substrings based on a delimiter (default is whitespace).

In [40]:
```
string = "apple,banana,orange"
fruits = string.split(',')
print(fruits)  # Output: ['apple', 'banana', 'orange']

string = "apple,banana,orange"
fruits = string.split('a')
print(fruits)  # Output: ['apple', 'banana', 'orange']
```

```
['apple', 'banana', 'orange']
['', 'pple,b', 'n', 'n', ',or', 'nge']
```

19.isalnum()

`isalnum()` : Checks if all characters in a string are alphanumeric (either alphabetic or numeric).

In [43]:
```
string = "Hello123"
print(string.isalnum())  # Output: True
```

```
True
```

20. isascii

`isascii()` : Checks if all characters in a string are ASCII characters.

In [46]:
```
string = "Hello"
print(string.isascii())  # Output: True
```

```
True
```

In [47]:
```
string = "Привет"
print(string.isascii())  # Output: False
```

```
False
```

String Module

The `string` module in Python provides a collection of useful constants and classes for working with strings. It is part of the Python Standard Library, which means it is available in all Python installations without the need for additional installation steps.

- Constants: The module includes several constants such as ascii_lowercase, ascii_uppercase, ascii_letters, digits, and punctuation. These constants provide predefined sets of characters that are commonly used in string manipulation tasks.
- Functions: The string module does not include standalone functions. However, you can use the constants provided in combination with other Python functions to perform various string operations.

In [49]:
```python
import string

# Accessing predefined sets of characters
print(string.ascii_uppercase)  # Output: ABCDEFGHIJKLMNOPQRSTUVWXYZ
print(string.ascii_lowercase)  # Output: abcdefghijklmnopqrstuvwxyz
print(string.digits)           # Output: 0123456789
print(string.punctuation)      # Output: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

Examples

Generating Random Strings

You can use the string.ascii_letters, string.digits, and other constants to generate random strings for tasks like generating passwords or tokens.

In [83]:
```python
import string
import random

# Generate a random password of length 8
password = ''.join(random.choices(string.ascii_letters + string.digits, k=8))
print(password)
```

```
n4Wf6TZ0
```

In [ ]:
```python
import string
import random
import hashlib

# Generate a opt of length 4
otp = ''.join(random.choices(string.digits, k=4))
print(otp)
result = (hashlib.sha512(otp.encode()))
```

```
print(result.hexdigest())
print(len(result.hexdigest()))
```

Validating User Input

You can use constants like string.digits to check if a user input contains only numeric characters.

In [86]:
```
import string

def is_numeric(input_string):
    for char in input_string:
        if char not in string.digits:
            return False
    return True

print(is_numeric("123"))  # Output: True
print(is_numeric("q123a")) # Output: False
```

```
True
False
```

In [87]:
```
import string

def is_numeric(input_string):
    return all(char in string.digits for char in input_string)

print(is_numeric("123"))  # Output: True
print(is_numeric("q123a")) # Output: False
```

```
True
False
```

Regular expressions

Regular expressions, often referred to as regex or regexp, are powerful tools for pattern matching and searching within strings. In Python, the re module provides support for working with regular expressions. Let's explore regular expressions with examples:

Basic Matching

In [90]:
```
import re

# Define a pattern
pattern = r'apple'

# Define a string to search
text = 'I have an apple and a banana.'

# Search for the pattern in the text
match = re.search(pattern, text)
```

```
if match:
    print('Pattern found:', match.group())
else:
    print('Pattern not found')
```

```
Pattern not found
```

In this example, re.search() is used to search for the pattern 'apple' within the text. If a match is found, match.group() returns the matched text.

Character Classes

In [ ]:
```
import re

# Define a pattern with character classes
pattern = r'[aeiou]'

# Define a string to search
text = 'I have an apple and a banana.'

# Find all vowels in the text
matches = re.findall(pattern, text)

print('Vowels found:', matches)
```

Here, the pattern [aeiou] matches any single vowel character in the text. re.findall() is used to find all occurrences of the pattern in the text.

Quantifiers

In [4]:
```
import re

# Define a pattern with quantifiers
pattern = r'\d{3}-\d{3}-\d{4}'

# Define a string to search
text = 'My phone number is 123-456-7890.'

# Search for a phone number pattern in the text
match = re.search(pattern, text)

if match:
    print('Phone number found:', match.group())
else:
    print('Phone number not found')
```

```
Phone number found: 123-456-7890
```

Groups

In [5]:
```
import re

# Define a pattern with groups
pattern = r'(\d{3})-(\d{3})-(\d{4})'
```

```
# Define a string to search
text = 'My phone number is 123-456-7890.'

# Extract area code, prefix, and line number
match = re.search(pattern, text)

if match:
    area_code, prefix, line_number = match.groups()
    print('Area Code:', area_code)
    print('Prefix:', prefix)
    print('Line Number:', line_number)
else:
    print('Phone number not found')
```

```
Area Code: 123
Prefix: 456
Line Number: 7890
```

Here, (\d{3})-(\d{3})-(\d{4}) defines three groups separated by hyphens, corresponding to the area code, prefix, and line number of a phone number. match.groups() returns a tuple containing the matched groups.

Any Questions?

For More

Refer the Lectures/Tutorials GitHub Page