NEHRU INSTITUTE OF TECHNOLOGY
AUTONOMOUS

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

x is positive
```

If-Else Statement

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

```
In [2]: y = -5
        if y > 0:
            print("y is positive")
        else:
            print("y is non-positive")
```

y is non-positive

Chained Conditional (If-Elif-Else)

Allows checking multiple conditions in sequence.

```
In [3]: z = 0
        if z > 0:
            print("z is positive")
        elif z < 0:
            print("z is negative")
        else:
            print("z is zero")
```

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

Count: 0
Count: 1
Count: 2
Count: 3
Count: 4

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

```
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

```
In [6]:  count = 5
         while count >= 0:
             print("Count:", count)
             count -= 1
```

```
Count: 5
Count: 4
Count: 3
Count: 2
Count: 1
Count: 0
```

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

```
apple
banana
cherry
```

```
In [8]:  for i in range(1,11):
             print(i)
```

```
1
2
3
4
5
6
7
8
9
10
```

Start Stop Step -3S in Range

In [13]:
```python
for i in range(1,10,1):
    print(i)
```

```
1
2
3
4
5
6
7
8
9
```

In [14]:
```python
for i in range(0, -10, -2):
    print(i)
```

```
0
-2
-4
-6
-8
```

Break Statement

Terminates the loop prematurely when a certain condition is met.

In [15]:
```python
for num in range(10):
    if num == 5:
        break
    print(num)
```

```
0
1
2
3
4
```

Continue Statement

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

```
In [16]:  for num in range(5):
              if num == 2:
                  continue
              print(num)
```

```
0
1
3
4
```

Pass Statement

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

```
In [17]:  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 [18]:  for c in "Hello, World":
              print(c)
```

```
H
e
l
l
o
,

W
o
r
l
d
```

String Slicing

```
In [19]:  my_str = "Python Lang"
          print(my_str[0])    # Output: 'P'
```

```
P
```

```
In [20]:  print(my_str[0:1])     # Output: 'P'

          P

In [21]:  print(my_str[0:5])     # Output: 'Pytho'

          Pytho

In [22]:  print(my_str[0:6])     # Output: 'Python'

          Python

In [23]:  print(my_str[6:7])     # Output: ' '


In [24]:  print(my_str[7:8])     # Output: 'L'

          L

In [25]:  print(my_str[-1:])     # Output: 'g'

          g

In [26]:  print(my_str[:])       # Output: 'Python Lang' (Full string)

          Python Lang

In [27]:  print(my_str[:6])      # Output: 'Python' (From the beginning up to index 6)

          Python

In [28]:  print(my_str[7:])      # Output: 'Lang' (From index 7 to the end)

          Lang

In [29]:  print(my_str[-4:])     # Output: 'Lang' (Last 4 characters)

          Lang

In [30]:  print(my_str[::2])     # Output: 'Pto ag' (Every other character) ->stepsize

          Pto ag

In [31]:  print(my_str[::-1])    # Output: 'gnaL nohtyP' (Reverse the string)

          gnaL nohtyP

In [32]:  # 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: !

          Characters in the string: Hello, World!
          First character: H
          Last character: !
```

```python
In [33]:  # 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
```

```
          ---------------------------------------------------------------------------
          TypeError                                 Traceback (most recent call last)
          Cell In[33], line 3
                1 # Example of string immutability
                2 # Attempting to modify a character in the string will result in an e
          rror
          ----> 3 my_string[0] = 'h'  # Raises TypeError: 'str' object does not suppor
          t item assignment

          TypeError: 'str' object does not support item assignment
```

```python
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)
```

```python
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)
```

```python
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

```python
In [34]:  my_string = "Hello, World!"
          print(len(my_string))
```

13

```python
In [35]:  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))
```

```python
    # 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"))
```

```
length of th given_str:  5
hi, World!
```

In [36]:
```python
"hello" + " " + "world"
```

Out[36]: `'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 [37]:
```python
string = "Hello"
print(len(string))   # Function Example: 5
```

```
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 [38]:
```python
string = "HELLO"
print(string.lower())  # Method Example: HELLO
```

```
hello
```

2. upper()

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

In [39]:
```python
string = "hello"
print(string.upper())  # Method Example: hello
```

```
HELLO
```

3. strip()

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

In [40]:
```python
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 [41]:
```python
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 [42]:
```python
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.

```python
In [43]:  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.

```python
In [44]:  string = "Hello, World!"
          print(string.startswith("Hello"))  # Method Example: True
```

```
True
```

8. endswith()

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

```python
In [45]:  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.

```python
In [46]:  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.

```python
In [47]:  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 [48]:  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 [49]:  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 [50]:  string = "123"
          print(string.isdigit())  # Method Example: True
```

True

14. islower()

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

```
In [51]:  string = "hell0"
          print(string.islower())  # Method Example: True
```

False

15. isupper()

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

```
In [52]:  string = "HELLO"
          print(string.isupper())  # Method Example: True
```

True

16. isspace()

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

```
In [53]:  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 [54]: 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 [55]: 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 [56]: string = "Hello123"
         print(string.isalnum())  # Output: True
```

True

20. isascii

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

```
In [57]: string = "Hello"
         print(string.isascii())  # Output: True
```

True

```
In [58]: 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 [59]: 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 [60]: import string
         import random

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

```
WtxTvX99
```

```
In [61]: 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()))
```

```
6778
4cf8a8a92e5ee1e70ad51372dc80f9d604b6a72fe952f7a1314fa7bce66c9332b834da8f922b
3bead2632165a4fd0762167d2d0548547b65e408ed8b5d3bf828
128
```

Validating User Input

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

In [62]:
```python
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 [63]:
```python
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 [64]:
```python
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 found: apple

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 [65]:
```
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)
```
Vowels found: ['a', 'e', 'a', 'a', 'e', 'a', 'a', 'a', 'a', 'a']

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 [66]:
```
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

```python
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?

- 1. Python Programming for Beginners: Skyrocket Your Code and Master Python in Less than a Week. Discover the Foolproof, Practical Route to Uncover Insider Hacks, Unlock New Opportunities, and Revolution Kindle Edition by Kit Jackson (Author), 31 May 2023
- 2. Python Programming for Beginners,ISBN-13-979-8870875248, Narry Prince, 2023

For More

Refer the Lectures/Tutorials GitHub Page