Python Tuple

Tuple is an ordered sequence of items same as a list. The only difference is that tuples are immutable. Tuples once created cannot be modified.

Tuples are used to write-protect data and are usually faster than lists as they cannot change dynamically.

It is defined within parentheses () where items are separated by commas.
We can use the slicing operator [ ] to extract items but we cannot change its value.

```
t = (5,'program', 1+3j)

# t[1] = 'program'
print("t[1] = ", t[1])

# t[0:3] = (5, 'program', (1+3j))
print("t[0:3] = ", t[0:3])

# Generates error
# Tuples are immutable
t[0] = 10
```

Python Set

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces {}. Items in a set are not ordered.

```
a = {5,2,3,1,4}

# printing set variable
print("a = ", a)

# data type of variable a
print(type(a))
```

We can perform set operations like union, intersection on two sets. Sets have unique values. They eliminate duplicates.

```
a = {1,2,2,3,3,3}

print(a)
```

Since, set are unordered collection, indexing has no meaning. Hence, the slicing operator [ ] does not work.

Python Dictionary

Dictionary is an unordered collection of key-value pairs.

It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data.

We must know the key to retrieve the value.

In Python, dictionaries are defined within braces {} with each item being a pair in the form key-value.

Key and value can be of any type.

```
d = {1:'value','key':2}
print(type(d))

print("d[1] = ", d[1])

print("d['key'] = ", d['key'])

# Generates error
print("d[2] = ", d[2])
```

Python if...else Statement

Decision making is required when we want to execute a code only if a certain condition is satisfied.

The if…elif…else statement is used in Python for decision making.

```
# If the number is positive, we print an appropriate message

num = 3

if num > 0:

    print(num, "is a positive number.")

print("This is always printed.")

num = -1

if num > 0:
```

```
    print(num, "is a positive number.")
print("This is also always printed.")
```

Example of if...else

```
# Program checks if the number is positive or negative
# And displays an appropriate message
num = 3
# Try these two variations as well.
# num = -5
# num = 0
if num >= 0:
    print("Positive or Zero")
else:
    print("Negative number")
```

Example of if...elif...else

```
'''In this program,
we check if the number is positive or
negative or zero and
display an appropriate message'''
num = 3.4
# Try these two variations as well:
# num = 0
# num = -4.5
if num > 0:
    print("Positive number")
```

```
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```

Python for Loop

Syntax of for Loop

```
For val in sequence:
        Loop body
```

Here, val is the variable that takes the value of the item inside the sequence on each iteration. Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Example:

```
# Program to find the sum of all numbers stored in a list

# List of numbers

numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum

sum = 0

# iterate over the list

for val in numbers:

    sum = sum+val

print("The sum is", sum)
```

The range() function

We can generate a sequence of numbers using range() function. Range(10) will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as range(start,stop,step). step_size defaults to 1 if not provided.

The range object is "lazy" in a sense because it doesn't generate every number that it "contains" when we create it. However, it is not an iterator since it supports in, len operations.

This function does not store all the values in memory; it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function list().

```
print(range(10))

print(list(range(10)))

print(list(range(2, 8)))

print(list(range(2, 20, 3)))
```

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate through a sequence using indexing. Here is an example.

```
# Program to iterate through a list using indexing

genre = ['pop', 'rock', 'jazz']

# iterate over the list using index

for i in range(len(genre)):

    print("I like", genre[i])
```

for loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.

The break keyword can be used to stop a for loop. In such cases, the else part is ignored.

Hence, a for loop's else part runs if no break occurs.

```
digits = [0, 1, 5]

for i in digits:

    print(i)

else:

    print("No items left.")
```

This for…else statement can be used with the break keyword to run the else block only when the break keyword was not executed.

```
# program to display student's marks from record

student_name = 'Yuvraj'

marks = {'Ashok': 90, 'Neeraj': 55, 'Jeetendra': 77}

for student in marks:

    if student == student_name:

        print(marks[student])

        break

else:

    print('No entry with that name found.')
```

Python while Loop

Example:

```
# Program to add natural

# numbers up to

# sum = 1+2+3+...+n

# To take input from the user,

# n = int(input("Enter n: "))

n = 10

# initialize sum and counter

sum = 0

i = 1

while i <= n:

    sum = sum + i

    i = i+1    # update counter

# print the sum

print("The sum is", sum)
```

While loop with else

Same as with for loops, while loops can also have an optional else block.
The else part is executed if the condition in the while loop evaluates to false.
The while loop can be terminated with a break statement. In such cases, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false.

```
'''Example to illustrate

the use of else statement

with the while loop'''
```

```
counter = 0
while counter < 3:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else")
```

Python break and continue

The working of break statement in for loop and while loop

```
# Use of break statement inside the loop
for val in "string":
    if val == "i":
        break
    print(val)
print("The end")
```

Python continue statement

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

The working of the `continue` statement in for and while loop

```
# Program to show the use of continue statement inside loops
for val in "string":
    if val == "i":
        continue
    print(val)
print("The end")
```