www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

FOURTH LECTURE – 01/10/25

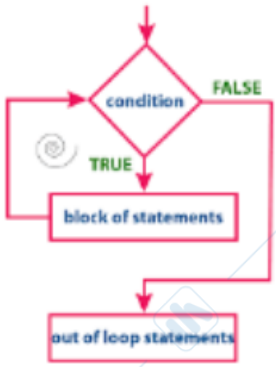
ITERATIVE PRIMITIVES

Iterative primitives, commonly known as loops, are used for executing a block of code repeatedly based on a condition. They provide a way to efficiently perform repetitive tasks in a program, thereby making it more concise and easier to maintain. Phyton offers two primary types of loops:

- While loop executes a block of code as long as a condition is true
- For loop executes a block of code for each item in a sequence

THE WHILE LOOP

The while loop repeatedly executes a block of code as long as a given condition remains true. This allows for flexible repetition based on dynamic conditions.




```
# Some code before the loop

while condition:
# code to execute while condition is true

#code after the loop
```

Note that:

- The loop will continue as long as the condition evaluates to True;
- It's crucial to include an "exit strategy" to eventually make the condition false, avoiding an infinite loop.

Simple example that uses a while loop to count from 1 to 5:

```
Count=1

While Count <= 5:

        Print(f"Current cpunt is {Count}")

        Count +=1

Print("Loop has ended")
```

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

The break keyword allows you to interrupt the normal execution of a loop and exit immediately. This is useful when you want to terminate a loop based on a specific condition that might occur during the loop's execution.

#some code before the loop

While some_condition:

    #some code

    If exit_condition:

        Break

    #some more code

#some code after the loop

WHILE TRUE LOOP

An infinite True loop can be useful when you want to keep running a block of code until a certain condition is met. The loop can be exited using the break keyword

```
while True:
    # some code
    if exit_condition:
        break
```

Example:

```
while True:
    user_input = input("Capitalizing your text. Type 'exit' to stop: ")
    if user_input == 'exit':
        break
    print(f"Capitalized text: {user_input.upper()}")
```

This program will keep asking for input and print it back until the user types 'exit'.

CONTINUE KEYWORD

The continue keyword allows you to skip the current iteration of a loop and continue with the next iteration. This is useful when you want to avoid certain computation or actions within a loop for a specific condition.

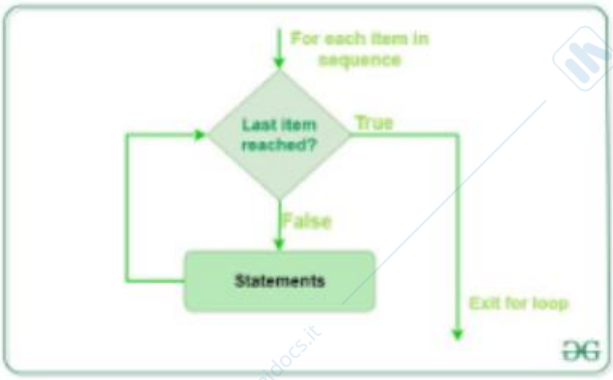
```
count = 1
upper = int(input('To which number do you want to count?'))

while count <= upper:
    if count == 13:
        count += 1 # why is this required?
        continue
    print(f"Current count is {count}")
    count += 1
```

The program counts from 1 up to the number specified by the user. The `continue` keyword allows it to skip the number 13 and continue counting.

THE FOR LOOP

The for loop is designed for scenarios where the number of iterations is known in advance or when we want to iterate over a sequence (like a list or string)

- Deterministic: for loops often have known number of iterations, making them more predictable
- Sequence iteration: for loops are naturally suited for iterating through sequences
- Simplex syntax: often requires less setup code than a while loop for the same tasks



www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Pros and cons:

- Easier to read and write for sequence-based loops
- Less prone to errors like infinite loops
- Less flexible for conditions that can't be determined in advance

The syntax of a for loop in Python is straightforward. It's designed to loop over a sequence and execute a block of code for each element in that sequence

```
for variable in sequence:
    # code to execute for each element
```

Here, sequence can be a list, tuple, string, or any iterable object. The variable takes on each value in the sequence one by one.

Here's a simple example of a for loop in Python:

```
fruits = ["Apple", "Banana", "Cherry"]
for fruit in fruits:
    print(f"I like {fruit}")
```

In this example, the for loop iterates through the list fruits, and the variable fruit takes on each value from the list. The output will be:

```
I like Apple
I like Banana
I like Cherry
```

In this example, the for loop counts from 1 up to the number specified by the user:

```
upper = int(input('To which number do you want to count?'))

for count in range(1, upper + 1):
    if count == 13:
        break
    print(f"Current count is {count}")
```

The program counts from 1 up to the number specified by the user. However, it exits the loop when count is equal to 13, thanks to the break keyword.

RANGE FUNCTION

The range function is commonly used with for loops to generate a sequence of numbers

Range (start, stop, step)

- Start: the starting number (optional, default is 0)
- Stop: the ending number (exclusive)
- Step: the step size (optional, default is 1)

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

For I in range (5):

Print(i)

ELSE SYNTAX

The else clause in a for or in a while loop executes only if the loop completes without a break

```
numbers = [2, 4, 6, 8]

for n in numbers:
    if n % 2 != 0:   # odd number found
        print("Found an odd number!")
        break
else:
    print("All numbers are even")
```

In this example, the loop checks if the list contains any odd number. Since the loop finishes without a break, the `else` clause executes and prints `"All numbers are even"`. The same works with `while...else`.

SUMMARIZING FOR AND WHILE LOOPS

A for loop provides a structured way to iterate a known number of times or over a known set of items. Underneath, it can be conceptually thought as a while loop with a counter.

FITH LECTURE – 02/10/25

STRING

A string can be considered a collection of characters. A string is an immutable sequence of characters, each of which can be letters, numbers, or symbols, enclosed within quotes.

String variable = “Hello world”
multi_line string= ‘’’This is a

Multi-line string’’’

In Python you can access individual characters of a string using indexing. It’s important to note that Python uses zero-based indexing, meaning the first character is at index 0 and not 1.

My_string=”Hello”

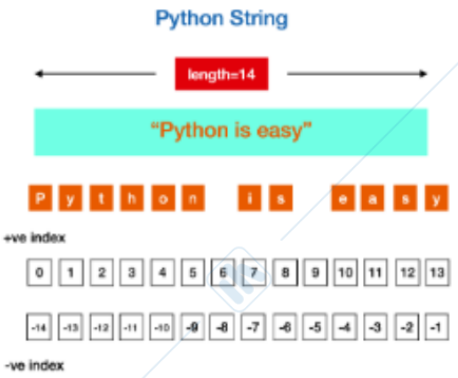
First_char=my_string[0] #It will be H

second_char=my_string[1] #It will be e

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

In python, strings are immutable, which means that once a string is created, its content cannot be changed directly.



In Python we can also have negative indexing for strings, which means you can count from the end of the string towards the beginning

THE LEN FUNCTION

The len function returns the length of a string.

String = “Hello, world!”

Length = len(string)

A string in Python is an iterable object, so it’s an object capable of returning its elements one at a time, permitting it to be iterated over in a loop

SLICING WITH STRINGS

A portion of a string is called slice. Slicing allows you to extract a substring from a given string by specifying the star and end indices.

Substring = string[start:end]

String = “Hello, world!”

Substring = string[7:12]

Print(substring) #output: world

There are some “special” cases for slicing:

- String[:4]: omits the start index and slices from the beginning up to index 3
- String[4:]: omits the end index and slices from index 4 to the end
- String[:]: omits both indeces and returns a copy of the whole string
- String[3:3]: a slice with the same start and end index returns an empty string

```
string = "Python"
print(string[:4])  # Output: "Pyth"
print(string[4:])  # Output: "on"
print(string[:])   # Output: "Python"
print(string[3:3]) # Output: ""
```

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

STRING METHODS

In Python, strings come with built-in ”capabilities” that allow you to perform various actions on them. These capabilities are known as string methods. Example:

string = “Python”

new_string = string.upper()

#new_string will now be “PYTHON”

Here are some of the most important methods:

- .upper(): converts all characters to uppercase
- .lower(): converts all characters to lower case
- .strip(): removes whitespace from the beginning to the end
- .replace(old, new): replaces occurrences of old with new
- .find(substring): finds the index of the first occurrence of substring
- .count(substring): counts occurrences of substring
- .split(delimiter): splits the string by delimiter into a list
- .join(list): joins a list of strings into a single string, separated by the string itself

NOTE: these methods do not modify the original string but produce a new one, in line with strings being immutable.

THE IN OPERATOR IN STRINGS

The in operator is a Boolean operator that allows you to check if a substring exists within a larger string case sensitive), returning True or False (for example to check if a password that you are creating contains a special character or a number)

```
if 'world' in 'Hello, world!':
    print("Substring found!") # output: Substring found!
```

STRINGS COMPARISON

Strings can be compared using ==, < and > operators. The comparison is done on the ASCII values of the characters.

```
# Equality
if string1 == string2:
    print("The strings are equal.")
# Less than
if string1 < string2:
    print(f"{string1} comes before {string2}.")
# Greater than
if string1 > string2:
    print(f"{string1} comes after {string2}.")
```

Here, "apple" comes before "banana" in lexicographical (ASCII) order.

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

LISTS

A list is a sequence of values. Unlike strings, which are sequences of characters, a list can contain values of any type. These values are called elements of the list.

Empty_list1=[]

Empty_list2=list()

List_of_numbers1=[1,2,3,4,5]

List_of_numbers2=list([1,2,3,4,5])

List_of_fruits=[“apple”,”banana”,”cherry”]

A list in python can contain a variety of types of values (numbers, strings, Booleans, even another list)

A list within another list is called a nested list

Lists in Python have serial key characteristics:

- Ordered: the elements in a list have a specific order that is maintained
- Mutable: the elements within a list can be changed
- Allow duplicates: lists can have elements that are repeated
- Indexed: you can access elements by their position in the list

```
# Create a list with duplicates
my_list = [1, 2, 2, 3]

# Access an element by index
print(my_list[1])  # Output: 2

# Modify an element
my_list[3] = 4
print(my_list)  # Output: [1, 2, 2, 4]
```

WARNING: while it’s possible to create lists with mixed types of elements, it’s generally better practice to keep lists homogenous.

Since lists are iterable objects, their traversal is very similar to the one of the strings (talking about for loop)

The + and * operators work with lists in a similar way they work with strings.

Concatenation:

```
list1 = [1, 2, 3]
list1 += [3, 4, 5]
print(list1) # Output: [1, 2, 3, 4, 5, 6]

list1 += 12 # Error
```

Repetition:

```
list1 = [1, 2, 3]
list1 *= 3
print(list1)# Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Slicing works with lists exactly like it does with strings. You can extract a sublist by specifying a start index and an end index

Special cases:

- `my_list[:4]` - from the start to index 3
- `my_list[3:]` - from index 3 to the end
- `my_list[:]` - a copy of the whole list
- `my_list[3:3]` - an empty list

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

LIST METHODS

Lists in Python have various built-in methods. Because lists are mutable, most of these methods modify the list directly and outputs None.

- .append(x): adds the element x to the end of the list
- .extend(L): appends all the elements in list L to the end
- .insert(i, x): inserts the element x at the given index i
- .remove(x): removes the first occurrence of the element x
- .pop(i): removes and returns the element at the given index i
- .index(x): returns the index of the first occurrence of the element x;
- .count(x): counts the occurrences of the element x;
- .sort(): sorts the list in ascending order (in-place);
- .reverse(): reverses the list (in-place).

EXAM SITUATION – APPEND VS EXTEND

Both append and extend can take a list as an argument, but they behave differently:

- Append: adds its entire argument as a single element to the end of the list. If the argument is a list, it will be added as a nested list
- Extend: adds each element in the given list argument to the list.

```
# Using append
my_list = [1, 2, 3]
my_list.append([4, 5])  # Result: [1, 2, 3, [4, 5]]

# Using extend
my_list = [1, 2, 3]
my_list.extend([4, 5])  # Result: [1, 2, 3, 4, 5]
```

Also in the list situation we can use the functions len() and sum():

- Len(): returns the number of elements in a list
- Sum(): adds up all the elements in a list of numbers

```
# Using len
my_list = [1, 2, 3, 4, 5]
length = len(my_list)  # Result: 5

# Using sum
my_list = [1, 2, 3, 4, 5]
total = sum(my_list)  # Result: 15
```

The .join() and .split() methods facilitate the conversion between lists and strings:

- .join(): combines a list of strings into a single string, with elements separated by a chosen delimiter
- .split(): breaks a string into a list of substrings based on a delimiter

```
# Using .join()
my_list = ['Hello', 'world']
joined_str = " ".join(my_list)  # Result: 'Hello world'

# Using .split()
my_str = 'apple, banana, cherry'
split_list = my_str.split(", ")
# Result: ['apple', 'banana', 'cherry']
```

LISTS: is VS ==

In Python, we have two ways to check for equality:

- ==: checks if the values are semantically equal
- Is: checks if two variables point to the same object in memory

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]
list3 = list1

print(list1 == list2)  # True
print(list1 is list2)  # False
print(list1 is list3)  # True
```

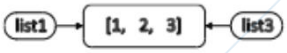
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

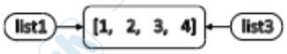
When two variables refer to the same list, we say that they are an alias of each other

Since list1 and list3 refer to the same object, changes in one are reflected in the other. Be cautious when working with alias as it might lead to unintended side-effects.

```
list1 = [1, 2, 3]
list3 = list1  # Alias
```




```
list3.append(4)
print(list1)  # Output: [1, 2, 3, 4]
```



For primitive data types like integers, floats, Booleans, NoneType and strings, the is and == operators often work similarly. This is because Python cache small integers and strings, so variables with the same value often point to the same object in memory.

LIST COMPREHENSIONS IN PYTHON

List comprehensions provide a concise way to create new lists based on existing ones or conditions. They are syntactic constructs that makes it easier to create collections based on existing ones.

Comprehensions make your code more Pythonic and easier to read.

```
# Without comprehension
squares = []
for i in range(1, 6):
    squares.append(i * i)

# With comprehension
squares = [i * i for i in range(1, 6)]

print(squares)  # Output: [1, 4, 9, 16, 25]
```

The concept of comprehension can be used also with strings. When a comprehension is applied to a string, the result is typically a list of characters or substrings, not a new string.

```
# Without comprehension
vowels = []
for char in 'hello':
    if char in 'aeiou':
        vowels.append(char)

# With comprehension
vowels = [char for char in 'hello' if char in 'aeiou']

print(vowels)  # Output: ['e', 'o']
```

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari