Lecture II - Control Structures for Your Code

Programming with Python

Dr. Tobias Vlćek
Kühne Logistics University Hamburg - Fall 2025

Quick Recap of the last Lecture

F-Strings

- F-strings provide a way to embed expressions inside string literals
- You can include expressions by placing them inside curly braces `{}`
- This makes it easier to include dynamic content

...

```
# Let's illustrate f-strings with a small example:
name = "Mr. Smith"
age = 30
height = 1.826549
print(f"My name is {name}, I'm {age} years old, and {height:.2f} meters
tall.")
```

```
My name is Mr. Smith, I'm 30 years old, and 1.83 meters tall.
```

...

> Tip
>
> We used the `:.2f` format specifier to round the number to two decimal places.

Variables and Data Types

- Python uses dynamic typing, i.e. the type is determined at runtime
- Basic data types in Python are: `int`, `float`, `str`, `bool`
- Variables are created by assignment with the `=` operator

...

> Question: What are the types of `y`, `z`, `w`?

```
y = 2.5
z = "Hello"
w = True
print(f"y is of type {type(y).__name__}")
```

```
print(f"z is of type {type(z).__name__}")
print(f"w is of type {type(w).__name__}")
```

```
y is of type float
z is of type str
w is of type bool
```

Arithmetic Operators

Addition
Subtraction
Multiplication
Division
Floor Division
Exponentiation
Modulo

`+`
`-`
`*`
`/`
`//`
`**`
`%`

Adds two numbers
Subtracts one number from another
Multiplies two numbers
Floating-point division
Integer division
Power of
Remainder of division

...

> **Note**
>
> Note, that the `/` operator always returns a float, even if the division is even. Furthermore, the `+` operator can be used to concatenate strings and that the `*` operator can be used to repeat strings.

Arithmetic Operators with Variables

- Additional operators can update the value of a variable (new)
- We can use `+=`, `-=`, `*=`, `/=`, `//=`, `**=`, `%=`

...

> Question: What is the value of `x` after the operations?

```python
x = 10
print(f"Initial value of x: {x}")
x += 5  # Equivalent to x = x + 5
print(f"After x += 5: {x}")
x *= 2  # Equivalent to x = x * 2
print(f"After x *= 2: {x}")
x %= 4  # Equivalent to x = x % 4
print(f"After x %= 4: {x}")
```

```
Initial value of x: 10
After x += 5: 15
After x *= 2: 30
After x %= 4: 2
```

Objects and Methods

Objects

- Objects are instances of classes
- We will learn more about classes later in the course
- In Python, virtually everything is an object
- Common built-in objects: integers, strings, lists, dictionaries
- For now, think of objects as a collection of data and methods

. . .

> **Note**
>
> For most programming purposes, you can treat everything in Python as an object. This means you can assign all types to variables, pass them to functions, and in many cases, call methods on them.

Methods

- Methods are functions that are called on an object
- The syntax is `object.method([arguments])`
- Methods are specific to the type of object they're called on
- They can modify the object or return information about it

. . .

> **Tip**
>
> You can use the `dir()` function to list all methods and attributes of an object.

String Methods

Here are some commonly used string methods:

- `upper()`: Converts all characters in the string to uppercase
- `lower()`: Converts all characters in the string to lowercase
- `title()`: Converts first character of each word to uppercase
- `strip()`: Removes leading and trailing whitespace
- `replace()`: Replaces a substring with another substring
- `find()`: Finds first substring and returns its index
- `count()`: Counts the number of occurrences of a substring

String Methods in Action

> Question: What will be the output of the following code?

```
message = "Hello, World!"
print(message.upper())  # Converts to uppercase
print(message.lower())  # Converts to lowercase
print(message.title())  # Converts to title case
print(message.replace("World", "Python"))  # Replaces "World" with "Python"
print(message.find("World"))  # Finds "World" and returns its index
print(message.count("o"))  # Counts the number of occurrences of "o"
```

```
HELLO, WORLD!
hello, world!
Hello, World!
Hello, Python!
7
2
```

...

> **i** Note
>
> Note, how `replace()` does not modify the original string. Instead, it returns a new string.

String Task

> Task: Discuss and implement the following task:

```
# Change the following message to get the desired output
message = " the snake programmer. "
# Your code here

output = "The Python Programmer."
```

...

> Tip
>
> Remember, that these methods return a new string. The original string is not modified.

String Task in Action

```
message = " the snake programmer. "
print(message.strip().title().replace("Snake", "Python"))
```

```
The Python Programmer.
```

...

> Tip
>
> Here we chained methods together to perform multiple operations after another in one line.

Indexing and Slicing

Indexing

- We have used indexing to access elements of a string last lecture
- It allows you to access elements of a sequence by position
- Positive indexing starts at 0 for the first element
- Negative indexing starts at –1 for the last element (new)

...

```
string_to_index = "Hello, World!"
print(string_to_index[0])  # Accessing the first character
print(string_to_index[-1]) # Accessing the last character
```

```
H
!
```

Slicing

- Slicing allows you to extract a portion of a sequence
- Syntax: `sequence[start:stop:step]`
- `start` is the index of the first element to include
- `stop` is the index of the first element to exclude
- `step` is the increment between indices (default is 1)
- The result is a new sequence containing the extracted elements

...

```
string_to_slice = "Hello, World!"
print(string_to_slice[7:12])   # Accessing the last five characters from
the start
print(string_to_slice[-6:-1])  # Accessing the last five characters from
the end
```

```
World
World
```

Slicing Simplified

- If we omit `start` or `stop`, it will be replaced by the start or end of the sequence, respectively
- If we omit `step`, it will be replaced by 1

...

```
string_to_slice = "Hello, World!"
print(string_to_slice[::2])   # Accessing every second character
print(string_to_slice[::-1])  # Accessing the string in reverse
```

```
Hlo ol!
!dlroW ,olleH
```

Slicing String Task

> Task: Discuss and implement the following task:

```
# Slice the following message to create the described output
message = "y6S0-teru89d23e'.n*ut"
# Your code here

output = "Student"
```

...

> Tip
>
> Remember, that these methods return a new string. The original string is not modified.

Comparisons

Comparison Operators

- Comparison operators are used to compare two values
- The result of a comparison is a boolean value (`True` or `False`)

...

> Question: What will be the output of the following code?

```
lower_number = 2; upper_number = 9
print(lower_number == upper_number) # Equality
print(lower_number != upper_number) # Inequality
print(lower_number > upper_number) # Greater than
print(lower_number < upper_number) # Less than
print(lower_number >= upper_number) # Greater than or equal to
print(lower_number <= upper_number) # Less than or equal to
```

```
False
True
False
True
False
True
```

Logical Operators

- Logical operators combine multiple comparison operators
- Common logical operators: `and`, `or`, `not`

...

> Question: Which of the following expressions is `True`?

```
lower_number = 2; middle_number = 5; upper_number = 9;
print(lower_number < middle_number and middle_number < upper_number) # and
print(lower_number < middle_number or middle_number > upper_number) # or
print(lower_number == lower_number and not lower_number > middle_number) #
not
```

```
True
True
True
```

...

> **i** Note
>
> Note, that `and` and `or` are evaluated from left to right.

Membership Operators

- Used to check if a value is present in a sequence
- Common membership operators: `in`, `not in`

...

> Question: Which of these expressions is `True`?

```python
an_apple = "apple"
print("a" in an_apple) # Check if "a" is in the string "apple"
print("pp" not in an_apple) # Check if "pp" is not in the string
```

```
True
False
```

...

> **Note**
>
> Note, that `in` and `not in` can be used for strings, lists, tuples, sets, and dictionaries. Don't worry! We will learn about lists, tuples, sets, and dictionaries later in the course.

Control Structures

Control Structures

- Used to control the flow of execution in a program
- They can be used to make decisions and repeat code blocks
- `if`, `elif`, `else`, `for`, `while`, `break`, `continue`

...

> Question: What do you think each of the above does?

Indentation

- Indentation is crucial in Python!
- It is used to indicate the block of code that belongs to the structure
- The standard indentation is 4 spaces
- You can use tabs, but you should be careful with that

...

> **Warning**
>
> Mixing tabs and spaces can cause errors that are difficult to debug. The Python style guide (PEP 8) recommends using 4 spaces per indentation level for consistency and readability.

Conditional Statements

Conditional Statements

- They are used to execute different blocks of code based on whether a condition is true or false:
  - `if` statements execute a block of code if a condition is `True`

- `elif` statements execute a block of code if the previous condition is `False` and the current condition is `True`
- `else` statements execute a block of code if the previous conditions are `False`

...

> Tip
>
> You can use the `and` and `or` operators to combine multiple conditions.

if-statements

```
condition = True
if condition:
    print("The condition is True!") # Code block to execute if condition is
True
print("This will always be printed!")
```

```
The condition is True!
This will always be printed!
```

...

```
condition = False
if condition:
    print("The condition is True!") # Code block to execute if condition is
True
print("This will always be printed!")
```

```
This will always be printed!
```

...

> Tip
>
> Writing `if condition:` is equivalent to `if condition == True:`

else-statements

```
condition = True
if condition:
    print("The condition is True!") # Code block to execute if condition is
True
else:
    print("The condition is False!") # Code block to execute if condition
is False
```

```
The condition is True!
```

...

```
condition = False
if condition:
    print("The condition is True!") # Code block to execute if condition is
True
else:
    print("The condition is False!") # Code block to execute if condition
is False
```

```
The condition is False!
```

elif-statements

```
temperature = 11
if temperature > 10:
    print("The temperature is greater than 10!")
elif temperature == 10:
    print("The temperature is equal to 10!")
else:
    print("The temperature is less than 10!")
```

```
The temperature is greater than 10!
```

...

```
temperature = 10
if temperature > 10:
    print("The temperature is greater than 10!")
elif temperature == 10:
    print("The temperature is equal to 10!")
else:
    print("The temperature is less than 10!")
```

```
The temperature is equal to 10!
```

Comparisons and Conditional Statements

> Question: What will be the output of the following code?

```
name = "Harry"
profession = "wizard"
age = 16
if name == "Harry" and profession == "wizard" and age < 18:
    print("You are the chosen one still visiting school!")
elif name == "Harry" and profession == "wizard" and age >= 18:
```

```
    print("You are the chosen one and can start your journey!")
else:
    print("You are not the chosen one!")
```

```
You are the chosen one still visiting school!
```

Loops

Loops

- Loops allow you to execute a block of code repeatedly
- There are two types of loops: `for` and `while`
- `for` loops are used to iterate over a sequence (e.g., list, tuple, string)
- `while` loops execute repeatedly until a condition is `False`

...

> Tip
>
> Nested control structures through further indentation are allowed as well, we thus can chain multiple control structures together.

for-loops

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

```
for i in range(0, 10, 2):
    print(i)
```

```
0
2
4
6
8
```

...

> Tip
>
> The `range()` function can take up to three arguments: start, stop, and step.

...

for-loops with Strings

> Question: What do you expect will be the output?

```
fruit = "yellow banana"
for letter in fruit:
    print(letter)
```

```
y
e
l
l
o
w

b
a
n
a
n
a
```

while-loops

```
i = 0
while i < 5:
    print(i)
    i += 1
```

```
0
1
2
3
4
```

...

> Question: What could be an issue with poorly written while-loops?

while True

> Question: Anybody an idea what this code does?

```
i = 0
while True:
```

```
    if i % 10 == 0:
        print(i)
    if i > 100:
        break
    i += 1
```

```
0
10
20
30
40
50
60
70
80
90
100
```

Importance of Control Flow

- Allows programs to make decisions based on conditions
- Enables repetition of code blocks
- Helps manage program complexity
- Improves efficiency by executing only necessary code
- Facilitates creation of dynamic, responsive programs

...

> **i** Note
>
> Without control flow, programs would execute linearly from top to bottom, limiting their functionality and flexibility.

Loop Task

> Task: Implement the following task:

```
# Implement a while-loop that prints all even numbers between 0 and 100
excluding both 0 and 100.
number = 0
# Your code here
```

...

> **i** Note
>
> And that's it for todays lecture!
> We now have covered the basics on String methods, Comparisons, conditional statements and loops.

Literature

Interesting Books to start

- Downey, A. B. (2024). Think Python: How to think like a computer scientist (Third edition). O'Reilly. Link to free online version
- Elter, S. (2021). Schrödinger programmiert Python: Das etwas andere Fachbuch (1. Auflage). Rheinwerk Verlag.

. . .

> Tip
>
> Nothing new here, but these are still great books to start with!

. . .

For more interesting literature to learn more about Python, take a look at the literature list of this course.