Lecture III - Building Reusable Functions

Programming with Python

Dr. Tobias Vlćek

Quick Recap of the last Lecture

Slicing

- With slicing we can get a range of elements from 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

. . .

> Tip
>
> If left out, the step defaults to 1. Else, start defaults to 0 and stop defaults to the length of the sequence. Negative indices can be used to slice from the end of the sequence.

Comparison Operators

- Comparison operators are used to compare two values
- The result of a comparison is a boolean value (`True` or `False`)
- Operators include: `==`, `!=`, `>`, `<`, `>=`, `<=`

. . .

> Question: Is this `True`?

```
# Careful here!
one = 1
two = 1
print(one == two)
```

```
True
```

Control Structures

- Control structures allow us to control the flow of execution
- It includes conditional statements and loops
- Conditional statements: `if`, `elif`, `else`
- Loops: `for` and `while`
- Control flow statements (in loops): `continue` and `break`

. . .

> **Note**
>
> The statement `continue` skips the rest of the current iteration and moves to the next one in a loop while the `break` statement exits the loop entirely.

Functions in Detail

What is a Function?

- Functions can accept inputs (parameters) and return outputs
- Encapsulate logic, making code easier to maintain
- Functions can be called **multiple times** from different part
- They help **reduce code duplication** and improve readability

```
# I'm a function.
type(print)
```

```
builtin_function_or_method
```

. . .

> ❗ Important
>
> Remember, methods are functions that are called on an object.

Some Built-in Functions already used

- `print()`: Print text to console
- `input()`: Read text from console
- `len()`: Get the length of a sequence
- `range()`: Generate a sequence of numbers
- `round()`: Round a number to a specified number of decimal places
- `type()`: Get the type of an object
- `int()`: Convert a string to an integer
- `float()`: Convert a string to a floating-point number
- `str()`: Convert an object to a string

Defining a Function

- Use the `def` keyword followed by the function name
- Inside parentheses we list the inputs (parameters)
- The code block within every function starts with a colon (:)
- It is indented, just as the loops from the last lecture

. . .

```
def greet(a_parameter):
    print(f"Hello, {a_parameter}!")
greet("Students")
```

```
Hello, Students!
```

...

> Tip
>
> It is common practice to leave out one line after the definition of a function, although we will not always do that in the lecture to save space on the slides.

Comment Functions

- It is good practice to include a comment at the top of your functions
- If you do it with three `"""`, it will appear in the help menu

...

```
def greet():
    """
    This function will be used later and has currently
    absolutely no use for anything.
    """
    pass # Necessary placeholder to avoid error

help(greet)
```

```
Help on function greet in module __main__:

greet()
    This function will be used later and has currently
    absolutely no use for anything.
```

Naming Functions (and Methods)

- Function names should be short, but descriptive
- Use underscores (_) instead of spaces in the names
- Avoid using Python keywords as function names (e.g., `print`)
- Try to avoid using built-in functions and methods that have a similar name (e.g., `sum` and `len`)

> Question: Which of the following is a good name for a function?

- `myfunctionthatmultipliesvalues`
- `multiply_two_values`

- `multiplyTwoValues`

Function Parameters

- Parameters are variables that the function accepts

- They allow you to pass data to the function
- Try to name them as variables: short and meaningful
- We can also **leave them out** or define **several inputs**!

```
def greet():
    print("Hello, stranger!")
greet()
```

```
Hello, stranger!
```

Function Arguments

- Arguments are the actual values passed to the function
- They replace the parameters in the function definition

...

> Question: What could be the correct arguments here?

```
def greet(university_name, lecture):
    print(f"Hello, students at the {university_name}!")
    print(f"You are in lecture {lecture}!")

# Your code here
```

Initializing Parameters

- We can also **initialize parameters** to a default value!
- To do this we use the = sign and provide it with a value
- This is called a keyword argument

```
def greet(lecture="Programming with Python"):
    print(f"You are in lecture '{lecture}'!")

greet()
greet("Super Advanced Programming with Python")
```

```
You are in lecture 'Programming with Python'!
You are in lecture 'Super Advanced Programming with Python'!
```

...

> Tip
>
> This is especially useful when we want to avoid errors due to missing arguments!

Multiple Parameters

- We can also have **multiple parameters** in a function definition
- They are called positional arguments and are separated by commas
- When we call them, they must be provided in the same order
- Alternatively, we could call them by name, as for example in this function call `print("h","i",sep='')`

. . .

> Question: What will be printed here?

```
def call_parameters(parameter_a, parameter_b):
    print(parameter_a, parameter_b)

call_parameters(parameter_b="Hello", parameter_a="World")
```

```
World Hello
```

Function Return Values

- Functions can return values using the `return` statement
- The return statement ends the function
- It then **returns** the specified value

. . .

```
def simple_multiplication(a,b):
    result = a*b
    return result
print(simple_multiplication(2,21))
```

```
42
```

. . .

```
def simple_multiplication(a,b):
    return a*b # even shorter!
print(simple_multiplication(2,21))
```

```
42
```

Access return values

- We can also **save** the return value from a function in a variable
- That way we can use it later on in the program

. . .

```
def simple_multiplication(a,b):
    return a*b # even shorter!

result = simple_multiplication(2,21)
print(result)
```

```
42
```

Returning None

- If we don't specify `return`, functions will return `None`

```
def simple_multiplication(a,b):
    result = a*b
```

```
print(simple_multiplication(2,21))
```

```
None
```

...

> Grouptask: Come up with a function that checks whether a number is positive or negative. It returns `"positive"` for positive numbers and `"negative"` for negative numbers.

...

> **Tip**
>
> You can also use **multiple return statements** in a function.

Scope

Function Scope

- Variables defined inside a function are local to that function
- They cannot be accessed outside the function

. . .

```
def greet(name):
    greeting = f"Hello, {name}!"

print(greeting)  # This will cause an error
```

. . .

> Question: Any idea how to access `greeting`?

Global Scope

- Variables defined **outside** all functions are in the global scope
- They can be accessed from anywhere in the program

. . .

```
greeting = "Hello, Stranger!"
def greet(name):
   greeting = f"Hello, {name}!"
   return greeting
print(greet("Students")) # Greet students
print(greeting) # Greet ????
```

```
Hello, Students!
Hello, Stranger!
```

. . .

> **!** Important
>
> We don't change global variables inside a function! The original value can still be accessed from outside the function.

Global Keyword

- Still, we can change the value of `greeting` from inside a function!
- By using the `global` keyword to modify a global variable

...

```
greeting = "Hello, Stranger!"

def greet(name):
   global greeting
   greeting = f"Hello, {name}!"
   return greeting

print(greet("Students")) # Greet students
print(greeting) # Greet students again
```

```
Hello, Students!
Hello, Students!
```

...

>Question: This can be confusing. Do you think you got the idea?

Classes

Classes

- Classes are blueprints for creating objects
- They encapsulate data (attributes) and behavior (methods)
- Objects are instances of classes
- Methods are functions that are defined within a class

...

```
class Students: # Class definition
    def know_answer(self): # Method definition
        print(f"They know the answer to all questions.")

student = Students() # Object instantiation
student.know_answer()
```

```
They know the answer to all questions.
```

Self

- Classes can be quite tricky at first, especially the `self` keyword
- When we call `self` in a method, it refers to the object itself
- It is used to **access** the attributes and methods of the class
- `self` always needs to be included in method definitions

...

```
# This won't work as self is missing
class Students: # Class definition
    def know_answer(): # Method definition without self
        print(f"They know the answer to all questions.")

student = Students()
student.know_answer()
```

...

>Task: Try it yourself, what is the error?

Naming Classes

- Classes can be named anything, but it is common to use the plural form of their name (e.g., `People`)

- CamelCase is used for class names, and snake_case is used for method and attribute names (e.g., `TallPeople`)
- Classes are usually defined in a file with the same name as their class, but with a `.py` extension

. . .

Question: Which of the following is a good class name? `smart_student`, `SmartStudent`, or `SmartStudents`

Class Attributes

- Class attributes are attributes that are shared by all class instances
- They are defined **within** the class but **outside** any methods

. . .

>Question: What do you think will happen here?

```
class Students: # Class definition
    smart = True # Class attribute

student_A = Students() # Object instantiation student_A
student_B = Students() # Object instantiation student_B

print(student_A.smart)
print(student_B.smart)
```

```
True
True
```

Instance Attributes

- Instance attributes are attributes unique to each class instance
- They are defined within the `__init__` method

```
class Student: # Class definition
    def __init__(self, name, is_smart): # Method for initalization
        self.name = name
        self.smart = is_smart
    def knows_answer(self): # Method to be called
        if self.smart:
            print(f"{self.name} knows the answer to the question.")
        else:
            print(f"{self.name} does not know the answer to the question.")

student = Student("Buddy",False) # Note, we don't need to call self here!
student.knows_answer()
```

```
Buddy does not know the answer to the question.
```

Inheritance

- Inheritance allows a class to inherit attributes and methods
- The class that inherits is called the **subclass**
- The class that is being inherited from is called the **superclass**

. . .

> Tip
>
> Don't worry! It can be quite much right now. Hang in there and soon it will get easier again!

Inheritance in Action

```
class Student: # Superclass
    def __init__(self, name):
        self.name = name
    def when_asked(self):
        pass

class SmartStudent(Student): # Subclass
    def when_asked(self):
        return f"{self.name} knows the answer!"

class LazyStudent(Student): # Subclass
    def when_asked(self):
        return f"{self.name} has to ask ChatGPT!"
```

>Grouptask: Create two students. One is smart and the other one is lazy. Make sure that both students reaction to a question is printed.

Encapsulation

- Encapsulation is the concept of bundling data (attributes) and methods (behavior) that operate on the data into a single unit (class)
- It is a key aspect of object oriented programming (OOP)
- It helps in organizing code and controlling access

. . .

> Note
>
> Fortunately, this is an introduction to Python, so we won't go into details of encapsulation.

The End

- Interested in more detail about classes and OOP?
- Check out access modifiers, getters and setters
- They are definitely a bit more complicated for beginners…
- Though they are worth learning if you build **complex programs**

. . .

> **Note**
>
> **And that's it for todays lecture!**
> We now have covered the basics of funtions and classes. We will continue with some slightly easier topics in the next lectures.

Literature {.title}

Interesting Book to dive deeper

- Thomas, D., & Hunt, A. (2019). The pragmatic programmer, 20th anniversary edition: Journey to mastery (Second edition). Addison-Wesley.

. . .

> **Tip**
>
> A fantastic textbook to understand the principles of modern software development and how to create effective software. Also available as a really good audiobook!

. . .

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