01b_python_basics

May 9, 2025

1 Python for Actuaries Part 1

1.1 Agenda

In this notebook, we will cover: - Definition and calling of functions - Parameters and return values - Introduction to lambda functions - Importing and using modules - Memory management in Python

1.2 Functions in Python

A function is a block-organized section of code that performs a specific task. Functions help to modularize the code, make it reusable, and improve clarity. They allow complex programs to be broken down into smaller, more manageable, and reusable parts.

By using functions, one can avoid redundancies in the code and make the development process more efficient. Instead of writing the same code multiple times, you define a function and call it as needed.

[ ]:
```
# simple function doing nothing
def foo():
    pass
```

[ ]:
```
# code which we write one time and use many times

# Definition of the function
def greet():
    print("Hi, welcome to Python!")

# function call
greet()
greet()
greet()
```

1.2.1 Parameters and Return Value

Functions become interesting only when we pass them something to work with and then process the return value `return`.

[ ]:
```
# simple with return value
def addition(a, b):
    return a + b
```

[ ]:
```
addition(1,1)
# Attention! No type checking in Python.
# Try addition("apples", "oranges") and see what happens.
```

[ ]:
```
import math

def quadratic_roots(a, b, c):
    disc = b**2-4*a*c
    if disc < 0:
        return None
    else:
        return (-b+math.sqrt(disc))/(2*a), (-b-math.sqrt(disc))/(2*a)
```

[ ]:
```
quadratic_roots(1, -5, 6) # eq = (x-3)(x-2) = x^2 -5x + 6
```

[ ]:
```
quadratic_roots(a=1, b=-5, c=6)
```

[ ]:
```
quadratic_roots(c=6, a=1, b=-5)
```

[ ]:
```
def create_character(name, race, hitpoints, ability):
    print('Name:', name)
    print('Race:', race)
    print('Hitpoints:', hitpoints)
    print('Ability:', ability)
```

[ ]:
```
create_character('Legolas', 'Elf', 100, 'Archery')
```

Standard values for transfer parameters

[ ]:
```
def create_character(name, race='Human', hitpoints=100, ability=None):
    print('Name:', name)
    print('Race:', race)
    print('Hitpoints:', hitpoints)
    if ability:
        print('Ability:', ability)
```

[ ]:
```
create_character('Jonas')
```

[ ]:
```
def create_character(name, race='Human', hitpoints=100, abilities=()):
    print('Name:', name)
    print('Race:', race)
    print('Hitpoints:', hitpoints)
    if abilities:
        print('Abilities:')
        for ability in abilities:
            print('  -', ability)
```

[ ]:
```
create_character('Gimli', race='Dwarf')
```

[ ]:
```
create_character('Gandalf', hitpoints=1000)
```

[ ]:
```
create_character('Aragorn', abilities=('Swording', 'Healing'))
```

Any number of parameters

[8]:
```
def create_character(name, *abilities, race='Human', hitpoints=100):
    print('Name:', name)
    print('Race:', race)
    print('Hitpoints:', hitpoints)
    if abilities:
        print('Abilities:')
        for ability in abilities:
            print('  -', ability)
```

[ ]:
```
create_character('Jonas')
```

[ ]:
```
create_character('Jonas', 'Coding', 'Teaching', 'Sleeping', hitpoints=25, )
```

2 Tasks

2.1 Task 1: Premium Calculation

Write a function `calculate_premium` that calculates the premium for an insurance policy. The function should accept the following parameters:

- `sum_insured`: The insured sum (in Euros)
- `risk_factor`: A risk factor (e.g., 1.2 for increased risk)
- `base_premium`: A base premium (in Euros)

The function should calculate the premium using the formula `premium = sum_insured * risk_factor * base_premium` and return it.

Example call:

```
premium = calculate_premium(100000, 1.2, 0.05)
print(f"The premium is: {premium} Euros")
```

[1]:
```
def calculate_premium(sum_insured, risk_factor, base_premium): # don't forget␣
↪the arguments
    return sum_insured * risk_factor * base_premium
```

[2]:
```
# Code for testing the function
calculate_premium(100000, 1.2, 0.05) # Ausgabe: 6000
```

[2]:
```
6000.0
```

2.2 Task 2: Claim Evaluation

Write a function `evaluate_claim` that evaluates a claim. The function should accept the following parameters:

- `claim_amount`: The amount of the damage (in Euros)
- `deductible`: The deductible (in Euros)

The function should calculate the payout amount by subtracting the deductible from the claim amount (if the claim amount is greater than the deductible) and return this value. If the claim amount is less than or equal to the deductible, the function should return `0`.

Example call:

```
payout = evaluate_claim(1500, 300)
print(f"The payout amount is: {payout} Euros")
```

[3]:
```
def evaluate_claim(claim_amount, deductible):
    return (max(0, claim_amount - deductible))
```

[4]:
```
# Code for testing
print(evaluate_claim(1500, 300)) # Ausgabe 1200
print(evaluate_claim(100, 300))  # Ausgabe 0
```

```
1200
0
```

2.3 Task 3: Age-Dependent Premium Adjustment

Write a function `adjust_premium_for_age` that adjusts the premium based on the age of the policyholder. The function should accept the following parameters:

- `base_premium`: The base premium (in Euros)
- `age`: The age of the policyholder

The function should use the following logic to adjust the premium:

- If the age is under 25 years, multiply the premium by 1.5.
- If the age is between 25 and 50 years, the premium remains the same.
- If the age is over 50 years, multiply the premium by 0.8.

The function should return the adjusted premium.

Example Call:

```
adjusted_premium = adjust_premium_for_age(100, 60)
print(f"The adjusted premium is: {adjusted_premium} Euros")
```

[ ]:
```
def adjust_premium_for_age(base_premium, age):
    if age < 25:
        premium = base_premium * 1.5
    elif age >= 25 and age < 50:
        premium = base_premium
    else:
        premium = base_premium * 0.8

    return premium
```

[ ]:
```python
# Code for testing
print(adjust_premium_for_age(100,23)) # 150
print(adjust_premium_for_age(100,49)) # 100
print(adjust_premium_for_age(100,50)) # 80
```

3 Functional Programming

Functional programming is a paradigm that focuses on computing values using functions. In Python, there are many built-in functions and concepts that are well-suited for functional programming, such as map(), filter(), reduce(), List Comprehensions (coming soon), and generators (which we will not cover).

Now let's take a look at anonymous functions (so-called lambda functions).

4 Lambda Functions

Lambda functions are small anonymous functions that are particularly useful when you want to use a simple function only once.

[ ]:
```python
# example
add = lambda x, y: x + y
print(add(3, 5))
```

[ ]:
```python
# Basisprämie und Risikofaktor
sum_insured = 100000  # versicherte Summe in Euro
risk_factor = 1.2     # Risikofaktor

# Lambda-Funktion zur Berechnung der Prämie
calculate_premium = lambda sum_insured, risk_factor: sum_insured * risk_factor␣
 ↪* 0.05

# Berechnung der Prämie
premium = calculate_premium(sum_insured, risk_factor)
print(f"Die berechnete Prämie beträgt: {premium:.2f} Euro")
```

5 Use of `import` in Python

In Python, the command `import` allows you to include modules in your program. A module is a file that contains Python code and defines functions, classes, and variables. By importing modules, you can access the functions and classes defined in those modules without having to rewrite them yourself.

5.1 Importing an Entire Module

You can import an entire module with the following command:

```
import modulname
```

[ ]:
```python
import math

# calculating the square root
root = math.sqrt(16)
print(f"The sqare root is: {root}")
```

Every now and then, you may not want to import the entire module but only specific functions. This can be done using

`from modulname import function`

[ ]:
```python
from math import sqrt

# calculating the square root
root = math.sqrt(25)
print(f"The sqare root is: {root}")
```

We will import many (significantly more powerful) modules/packages during the course of the event.

5.1.1 Memory Management in Python

Memory management in Python is largely automatic, thanks to a mechanism known as **Garbage Collection**. Nevertheless, there are some fundamental concepts that are important for understanding efficient work with memory resources.

Important Concepts of Memory Management in Python

1. **Memory Allocation**:
   - When you assign a value to a variable, Python automatically creates an object in memory that contains this value, and the variable points to it.
   - Python distinguishes between **mutable** (changeable) and **immutable** (unchangeable) objects:
     - Mutable objects (like lists or dictionaries) can be modified after assignment.
     - Immutable objects (like strings, tuples, and numbers) cannot be changed. For example, if you assign a new value to an existing variable, Python creates a new object and assigns the new memory address to the variable.
2. **Reference Counting**:
   - Python uses a **reference counting system** to track how many variables point to a specific object. When the count for an object drops to **0** (meaning there are no more references to it), the memory is automatically freed.
3. **Garbage Collection**:
   - Python has an automatic **Garbage Collection** that cleans up memory space that is no longer needed. It primarily relies on the reference counting system to decide when objects should be deleted.
4. **Memory Release with `del`**:
   - With the **`del` statement**, you can explicitly delete the reference to an object, which reduces its reference count. If there are no further references to the object, it is removed from memory.

[ ]:
```python
# Example 1: Mutable and Immutable Objects
print("Mutable vs Immutable Objects")

# Immutable: Strings
a = "Hallo"
print(f"Before change: a = {a} (ID: {id(a)})")
a = "Welt"  # A new object is created
print(f"After change: a = {a} (ID: {id(a)})")

# Mutable: Lists
b = [1, 2, 3]
print(f"\nBefore change: b = {b} (ID: {id(b)})")
b.append(4)  # The list is modified in memory
print(f"After change: b = {b} (ID: {id(b)})")

# Example 2: Reference counting and `del`
print("\nReference counting and memory management")

x = [1, 2, 3]
y = x  # x and y refer to the same object
print(f"Reference count for x and y: {id(x)} == {id(y)}")

del x  # Deletes the reference x
print("x deleted. Remaining access to the object via y:", y)

# Garbage collection occurs when no references remain
```