Functions, Lambda, Decorators, Generators, Type Hints

Functions

Calling a built-in function

In [1]:
```
print("hello")
```

```
hello
```

Calling a function with multiple arguments

In [2]:
```
name = "abcd"
print("hellow", name)
```

```
hellow abcd
```

In [3]:
```
ls1 = [1, 2, 3]
```

Calling a function with named arguments

In [4]:
```
for x in ls1:
    print(x**2, end=" ")
```

```
1 4 9
```

In [5]:
```
for x in ls1:
    print(x**2)
```

```
1
4
9
```

Defining a function

Functions are defined with the def keyword followed by the function name and its arguments in parentheses. Note that, as always, codes inside the function block need to be indented.

In [6]:
```
def add_two_things(a, b):
    return a+b
```

In [7]:
```
add_two_things(5, 6)
```

Out[7]:
```
11
```

In [8]:
```
result = add_two_things(5, 6)
result
```

Out[8]: 11

In [9]:
```
print(add_two_things(5, 6))
```

11

In [10]:
```
add_two_things("hello", " world")
```

Out[10]: 'hello world'

In [11]:
```
add_two_things([5, 6], [7, 8])
```

Out[11]: [5, 6, 7, 8]

In [12]:
```
# add_two_things(5, "abc")    # this will throw an error: TypeError: unsupported
```

The above function does not care about the types of arguments that are passed. It will try to "add" them regardless of the type. A rudimentary way of enforcing types in a function can be as follows:

(Note: `isinstance(object_A, type_B)` checks whether `object_A` is of `type_B`)

In [13]:
```
def add_two_integers(a, b):
    if isinstance(a, int) and isinstance(b, int):
        return a + b
    else:
        return "Please provide two integers"
```

Example: factorial of an integer, n

In [14]:
```
def fact(n):
    result = 1
    while n>1:
        result *= n
        n -= 1
    return result
```

In [15]:
```
fact(5)
```

Out[15]: 120

Example: permutation, ${}^nP_r$

In [16]:
```
def permutation(n, r):
    return fact(n)/fact(n-r)
```

In [17]:
```
permutation(5, 3)
```

Out[17]: 60.0

We can provide the arguments to the function with their names:

In [18]:
```
permutation(n = 5, r = 3)
```

Out[18]: 60.0

When providing the arguments with their names, their positions can be altered.

In [19]:
```
permutation(r = 3, n = 5)
```

Out[19]: 60.0

Example: GCD function

In [20]:
```
def gcd(a,b):
    if a<b:
        gcd=a
    else :
        gcd=b
    while a%gcd!=0 or b%gcd!=0 :
        gcd-=1
    return gcd
```

In [21]:
```
gcd(25,15)
```

Out[21]: 5

Recursive function

A function is recursive when it calls itself inside its codeblock. Since the function calls itself, a base case which - when reached - terminates the function is a must, otherwise, the function will turn into an infinite loop.

For example, the factorial of a number can be thought of as a recursive function:

$$f(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \times f(n-1), & \text{if } n > 0 \end{cases}$$

Here, the case in which n equals to 0, is the base case. Upon reaching n = 0, the function will terminate.

In [22]:
```
def fact_recurs(n):
    if n==0:
        return 1
    else:
        return n * fact_recurs(n-1)
```

In [23]:
```
fact_recurs(5)
```

Out[23]: 120

Default Values for Arguments

Functions can have default values for arguments. When a function has a default value for one or more argument(s), we can omit them when calling that function.

For example, the default value for the `end` argument in the `print()` function is set to `"\n"`.

In [24]:
```
print("Hellow", end = "\n")
print("world")
```

```
Hellow
world
```

In [25]:
```
print("Hellow")
print("world")
```

```
Hellow
world
```

In [26]:
```
print("Hellow", end = " ")
print("world")
```

```
Hellow world
```

Defining functions with default values for arguments

In [27]:
```
def power(base, exponent = 2):
    return base**exponent
```

In [28]:
```
power(5)
```

Out[28]: 25

In [29]:
```
power(6, 3)
```

Out[29]: 216

Lambda functions

Lambda functions in Python are **small, anonymous functions** — meaning they are defined **without a name**.

Used when you need a simple function for a short period of time. Usually used inside other functions.

In other programming languages, similar concepts exist under different names:

- **JavaScript:** Arrow functions (`x => x + 1`)
- **C++ / Java:** Lambdas (`[](int x){ return x + 1; }`)
- **R / MATLAB:** Anonymous functions
- **Haskell:** Lambda expressions (the origin of the term)

Syntax and Basic Definition

Lambda functions are defined using the `lambda` keyword followed by the arguments and the one-line function body.

Lambda functions are best for **small, throwaway** operations. For **larger logic or reuse**, use a named function (`def`). Use `lambda` for **short, one-liner functions**, and `def` for **anything more complex**.

In [30]:
```
# below we define a lambda function that squares its input
# then assign it to the variable name square.
square = lambda x: x**2

# note that, lambda functions are usually not assigned names like this.
# it is for demonstration purpose only.

# also note, functions in python are "first-class objects", meaning
# that functions can be assigned names, passed to other functions etc.
# more on this in upcoming objected-oriented programming lectures
```

In [31]:
```
square(2)
```

Out[31]:
```
4
```

Lambda with Multiple Arguments

Lambdas can have multiple arguments

In [32]:
```
# Lambda with two arguments
add = lambda a, b: a + b
add(3, 7)
```

Out[32]:
```
10
```

In [33]:
```
# Lambda to compare two values
max_val = lambda x, y: x if x > y else y
max_val(8, 3)
```

Out[33]:
```
8
```

In [34]:
```
# Lambda for string formatting
format_name = lambda first, last: f"{last}, {first}"
print(format_name("Alan", "Turing"))
```

```
Turing, Alan
```

Common Use Cases in Python

Lambda functions are often used with Python's higher-order functions — functions that take other functions as arguments.

a. `map()` — Apply a function to each element

In [35]:
```
# List of numbers
nums = list(range(1, 11))

# Square each number using map + lambda
squared = list(map(lambda x: x ** 2, nums))
squared
```

Out[35]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

In [36]:
```
words = ["python", "java", "perl", "javascript", "php"]
upper_words = list(map(lambda w: w.upper(), words))
print(upper_words)
```

```
['PYTHON', 'JAVA', 'PERL', 'JAVASCRIPT', 'PHP']
```

b. `filter()` — Keep elements that satisfy a condition

In [37]:
```
# Keep only even numbers
evens = list(filter(lambda x: x % 2 == 0, nums))
evens
```

Out[37]:
```
[2, 4, 6, 8, 10]
```

c. `sorted()` — Custom sorting using a key function

In [38]:
```
# List of tuples (name, age)
people = [{"name" : "Alice", "age" :30}, {"name" : "Bob", "age" : 25}, {"name" :

# Sort by age using lambda
sorted_people = sorted(people, key=lambda person: person["age"], reverse=True)
sorted_people
```

Out[38]:
```
[{'name': 'Eve', 'age': 35},
 {'name': 'Alice', 'age': 30},
 {'name': 'Bob', 'age': 25}]
```

In [39]:
```
def sorter(person):
    return person["age"]

sorted(people, key=sorter, reverse=True)
```

Out[39]:
```
[{'name': 'Eve', 'age': 35},
 {'name': 'Alice', 'age': 30},
 {'name': 'Bob', 'age': 25}]
```

In [40]:
```
# sorted in descending order
desc_sorted_people = sorted(people, key=lambda person: -person["age"])
desc_sorted_people
```

Out[40]:
```
[{'name': 'Eve', 'age': 35},
 {'name': 'Alice', 'age': 30},
 {'name': 'Bob', 'age': 25}]
```

d. `reduce()` — Combine elements cumulatively

In [41]:
```
from functools import reduce
```

```
# Sum of list using reduce + lambda
nums = [1, 2, 3, 4, 5]
total = reduce(lambda x, y: x + y, nums)

total
```

Out[41]: 15

In [42]:
```
nums = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, nums)
print(product)
```

```
24
```

In [43]:
```
import functools
```

In [44]:
```
functools.reduce(lambda x, y: x + y, nums)
```

Out[44]: 10

Type Hints in Python

Python is a dynamically typed language. We do not have to mention the types of objects when defining varibales or function arguments. While this makes the code concise, it can also lead to typing related bugs and confusions.

Type hints, as the name suggests, are a way to hint to programmers and Python about the types of objects.

To reduce runtime surprises, **Type Hints** were introduced in **Python 3.5 (PEP 484)**.

Type hints (or **type annotations**) allows one to explicitly state what type a variable or function parameter should be, without changing how Python executes the code.

NOTE: Python **ignores** type hints at runtime. They are mainly for **developers, IDEs, and tools** like `mypy` or `pyright`.

Why use type-hints:

- **Clarity**: Makes it clear what type of data is expected.
- **Readability**: Easier for others (and your future self) to understand the code.
- **Static Checking**: Tools like `mypy` can detect mismatched types before runtime.
- **Fewer Bugs**: Early detection of type-related issues.
- **IDE Support**: Better autocomplete and type-aware refactoring.

Annotating Variables

In [45]:
```
name: str = "Alice"
age: int = 25
height: float = 5.6
```

Annotating Functions

In [46]:
```python
def multiply(x: int, y: int) -> int:
    return x * y
```

Functions with Multiple Types

In [47]:
```python
from typing import Union

def add(a: Union[int, float], b: Union[int, float]) -> Union[int, float]:
    return a + b
```

In [48]:
```python
# the greet function takes an argument which is a string and returns a string
def greet(name: str) -> str:
    return f"Hello, {name}!"
```

Hints for collections and optional values

In [49]:
```python
from typing import List, Dict, Tuple, Set, Union, Optional

numbers: List[int] = [1, 2, 3]
person: Dict[str, Union[str, int]] = {"name": "Alice", "age": 25}
coordinates: Tuple[int, int] = (10, 20)
tags: Set[str] = {"python", "typing"}
nickname: Optional[str] = None  # Optional[str] means either str or None
```

In [50]:
```python
numbers= [1, 2, 3]
person = {"name": "Alice", "age": 25}
coordinates = (10, 20)
tags = {"python", "typing"}
nickname = None  # Optional[str] means either str or None
```

Type hints are **not enforced** by Python — but you can use external tools to check them. For example, `mypy` is a static type checker for Python.

Example:

```
def add(a: int, b: int) -> int:
    return a + b

result = add(5, "3")  # type checker will flag this before
runtime
```

If the above code is run, python will throw an error because `int` and `str` can not be added. However, using a type checker *before* running the code would allow one to detect this bug early