Machine Learning

Python Programming Basics & Operators

Functions & Modular Programming

- Defining functions
- Parameters & return values
- Reusability
- Introduction to modules

Learning Objectives

By the end of this session students should be able to:

- Define and call Python functions
- Use positional, keyword, and default parameters
- Return single and multiple values; understand scope
- Write simple recursive functions with proper base cases
- Create, import, reload, and run modules (.py files) from the working folder

Why Functions & Modules?

- Functions encapsulate behavior: avoid repetition (DRY principle)
- Modules (.py files) package related code for reuse across projects
- Benefits: easier testing, clearer code, better maintainability

Parts of a Function

Part	Purpose	Example
`def`	Keyword to define a function	`def greet():`
`function_name`	Name of function (snake_case)	`def calculate_total():`
`parameters`	Inputs to the function	`(name, age)`
`docstring`	Description of what it does	`"""Return greeting string."""`
`return`	Send value back to caller	`return greeting`

Function Basics — Syntax

```
def function_name(param1, param2=default):
    """Optional docstring describing behavior."""
    # body
    return result  # optional
```

Example:

```
def greet(name):
    """Return a greeting string."""
    return f"Hello, {name}!"
```

Calling Functions — Simple Example

```
print(greet("Sita"))      # Hello, Sita!
```

Step-through:

- Python evaluates arguments, calls the function, executes body and returns result.

Parameters & Argument Types

- Positional arguments: order matters
- Keyword arguments: name=value (order independent)
- Default values: make arguments optional
- Variable-length: *args (positional), **kwargs (keyword)

Example:

```
def power(x, p=2):
    return x ** p

power(3)        # 9
power(3, p=3)   # 27
```

Returning Values

- Use return to send values back to the caller
- Multiple values returned as a tuple (can be unpacked)

Example:

```
def min_max(values):
    return min(values), max(values)

lo, hi = min_max([4, 1, 9])
print(lo, hi)  # 1 9
```

Variable Scope — Local vs Global

- Local: variables defined inside a function (not visible outside)
- Global: variables at module level (visible across the module)
- Prefer returning values instead of mutating globals

Example:

```
x = 10
def foo():
    x = 5
    return x

print(foo(), x)  # 5 10
```

Recursion — Concept & Example

- Recursion: a function calls itself to solve a smaller subproblem
- Always include a base case to stop recursion

Example (factorial):

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)
```

Troubleshooting

Common Errors

Error	Cause	Solution
`NameError: name 'func' is not defined`	Function called before definition	Define function before using it
`TypeError: func() missing 1 required positional argument`	Wrong number of arguments	Check function signature
`ModuleNotFoundError: No module named 'xyz'`	Module file not found	Ensure .py file is in same folder
`RecursionError: maximum recursion depth exceeded`	Missing/wrong base case	Add proper base case to recursion
`AttributeError: module has no attribute 'func'`	Function not in module	Check module has the function

Debugging Tips

```python
# 1. Print variable values
def mysterious_function(x):
    print(f"Input: {x}")
    result = x * 2
    print(f"Result: {result}")
    return result

# 2. Check function exists in module
import datautils
print(dir(datautils))  # List all functions/variables

# 3. Verify imports
print(f"Imported module location: {datautils.__file__}")

# 4. Use assertions to test
def add(a, b):
    return a + b

assert add(2, 3) == 5, "add function not working"
print("Test passed!")
```

Modules — What & Why

- A module is a .py file containing functions, classes, and variables
- Import modules into other scripts or interactive sessions to reuse functionality
- Import styles:
  - import module → use module.name
  - from module import name → use name directly

Create a Module File — Simple Approaches

A. Create using a text editor and save as mymodule.py in the project folder (recommended).
B. Programmatic write from Python (useful for quick examples):

```
code = """def greet(name):
    return f'Hello, {name}!'
"""
open("mymodule.py", "w").write(code)
```

After saving the file in the same folder as your code, it can be imported as a module.

Example Module — datautils.py

```
def first_n_squares(n):
    return [i*i for i in range(1, n+1)]

def is_prime(n):
    if n <= 1:
        return False
    if n % 2 == 0 and n != 2:
        return False
    i = 3
    while i * i <= n:
        if n % i == 0:
            return False
        i += 2
    return True
```

```python
def bmi_info(weight_kg, height_m):
    """Calculate BMI and classify health category.

    Args:
        weight_kg (float): Weight in kilograms
        height_m (float): Height in meters

    Returns:
        tuple: (bmi, category)
    """
    bmi = weight_kg / (height_m ** 2)
    if bmi < 18.5:
        cat = "Underweight"
    elif bmi < 25:
        cat = "Normal"
    elif bmi < 30:
        cat = "Overweight"
    else:
        cat = "Obese"
    return bmi, cat

if __name__ == "__main__":
    # quick checks when the file is executed directly
    print("First 5 squares:", first_n_squares(5))
    print("Primes 1..20:", [n for n in range(1,21) if is_prime(n)])
    print("BMI example (70kg,1.75m):", bmi_info(70, 1.75))
```

Import and Use the Module

In another Python file or session within the same folder:

```
import datautils

print(datautils.first_n_squares(5))
print([n for n in range(1,21) if datautils.is_prime(n)])
print(datautils.bmi_info(70, 1.75))
```

Or import specific names:

```
from datautils import is_prime, bmi_info
print(is_prime(17))
print(bmi_info(60, 1.6))
```

Updating a Module During Development

- When you edit a module during the same interpreter/session, reload it:

```
import importlib
importlib.reload(datautils)
```

- Alternatively, start a fresh interpreter/session to get a clean import.

Organizing Code — Simple Project Layout

```
project/
  datautils.py
  scripts/
    run_demo.py
  README.txt
```

Keep reusable modules at top-level of the project so other files can import them directly.

Best Practices & Common Pitfalls

- Keep functions focused and small
- Add docstrings and short comments for public functions
- Avoid heavy computation at import time (use demo/test blocks)
- Prefer explicit imports (`import module`) over `from module import name`
- For mutable default arguments use None and initialize inside the function

Example (safe default pattern):

```
def append_item(x, lst=None):
    if lst is None:
        lst = []
    lst.append(x)
    return lst
```

Hands-on Exercises (in-class / lab)

1. Create `datautils.py` with the example functions. Run the file's quick checks to verify outputs.
2. In a separate file, import `datautils` and call each function for sample inputs.
3. Improve `is_prime` for small optimizations (handle even numbers, skip even divisors).
4. Add simple assertion tests in a small test file to validate behavior.

Assessment / Homework

- Deliverables:
  - `datautils.py` with required functions and docstrings
  - a short script showing how you import and use the module (e.g., `scripts/run_demo.py`)
  - `README.txt` listing exact steps you used to run the demo in your environment

Evaluation will check correctness, code clarity, and presence of basic tests.