# Functions in Python

Martin Wafula
Multimedia University of Kenya

# Outline

Introduction to Functions

Function Syntax

Parameters and Arguments

Return Values and Scope

Advanced Function Concepts

Best Practices

Practical Examples

Conclusion

Python Modules and Packages

# Introduction to Functions

# What Are Functions?

- A function is a reusable block of code that performs a specific task
- Functions help in:
  - Code organization and modularity
  - Reducing code duplication
  - Improving readability and maintainability
- Mathematical analogy: $f(x) = x^2$ vs Python's `def square(x): return x*x`

# Function Characteristics

- Has a **name** (should be descriptive and follow naming conventions)
- Takes **parameters** (0 or more)
- Has a **docstring** (optional but recommended)
- Contains a **body** with the implementation
- May **return** a value (or `None` if no return statement)

# Function Syntax

# Function Definition

## Basic Syntax

```
def function_name(parameters):
    """docstring - describes what the function does"""
    # Function body
    # ...
    return value  # optional
```

## Example

```
def greet(name):
    """Prints a greeting message"""
    print(f"Hello, {name}!")
    return len(name)  # returns length of name
```

# Function Invocation

## Calling a Function

```
result = function_name(arguments)
```

## Example

```
>>> message_length = greet("Alice")
Hello, Alice!
>>> print(message_length)
5
```

# Parameters and Arguments

# Parameters vs. Arguments

- **Parameters**: Variables in the function definition
- **Arguments**: Actual values passed to the function
- Python supports several argument types:
    - **Positional arguments**: most basic types of arguments. They are passed **in order** to the function and match the function parameters.
    - **Keyword arguments**: Are passed using parameter names rather than position. Order doesn't matter.
    - **Default arguments**: Have a default value if no value is provided. They must come after required (non-default) parameters.
    - **Variable-length arguments (*args and **kwargs)**: We use them when we do not know the number of arguments

# Positional vs Keyword Arguments

## Positional

```
def describe_pet(animal, name):
    print(f"I have a {animal} named {name}")

describe_pet("dog", "Rex")
```

## Keyword

```
def describe_pet(animal, name):
    print(f"I have a {animal} named {name}")

describe_pet(name="Rex", animal="dog")
```

# Default Arguments

## Function Definition

```
def describe_pet(name, animal="dog"):
    print(f"I have a {animal} named {name}")
```

## Function Calls

```
describe_pet("Rex")          # Uses default
describe_pet("Fluffy", "cat")  # Overrides default
```

# Variable-Length Arguments

*args (allows function to accept any number of positional arguments. Inside function, args is a tuple)

```
def average(*numbers):
    return sum(numbers)/len(numbers)

print(average(1, 2, 3))  # 2.0
```

**kwargs (allows a function accept any number of key arguments or named arguments. kwargs is a dictionary.)

```
def person_info(**details):
    for key, value in details.items():
        print(f"{key}: {value}")

person_info(name="Alice", age=25, city = "London")
```

# Differences between *args and **kwargs

| Feature | *args | **kwargs |
|---|---|---|
| Type | Tuple | Dictionary |
| Usage | Collects positional arguments | Collects keyword ar |
| Example call | func(1, 2, 3) | func(a = 1, b = |
| Unpacking | *args unpacks a tuple/ list | **kwargs unpacks a d |

# Return Values and Scope

# Return Statement

- The `return` statement exits the function and optionally returns a value
- Functions without `return` return `None`
- Multiple values can be returned as a tuple

### Example

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

low, high = min_max([3, 1, 4, 1, 5, 9])
```

# Variable Scope

- **Local variables**: Defined in a function, only accessible within it
- **Global variables**: Defined outside functions, accessible throughout
- The `global` keyword allows modifying global variables inside functions

Example

```
def increment():
global count
count += 1
```

# Advanced Function Concepts

# Lambda Functions

- Anonymous functions defined with `lambda`
- Typically used for short, simple operations
- Syntax: `lambda parameters: expression`

Example

```
square = lambda x: x * x
sorted_names = sorted(names, key=lambda x: x.lower())
```

# Function Decorators

- Functions that modify the behavior of other functions
- Useful for adding functionality without changing the original code

## Example

```
def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(f"Time: {time.time()-start}")
        return result
    return wrapper

@timer
def compute():
    # Long computation
```

# Recursion

- A function that calls itself
- Must have a base case to prevent infinite recursion

Example

```
def factorial(n):
    if n == 0:  # Base case
        return 1
    return n * factorial(n-1)
```

# Best Practices

# Function Design Guidelines

- **Single Responsibility Principle**: Each function should do one thing
- Use descriptive names (verbs for actions, nouns for data)
- Limit parameters (3-4 max)
- Use default arguments for common cases
- Document with docstrings
- Avoid modifying mutable arguments unless intended
- Prefer returning values over modifying globals

# Effective Docstrings

## Good Documentation

```
def calculate_area(length, width):
    """
    Calculate the area of a rectangle.
    Args:
        length (float): The length of the rectangle
        width (float): The width of the rectangle
    Returns:
        float: The area of the rectangle
    Raises:
        ValueError: If either dimension is negative
    """
    if length < 0 or width < 0:
        raise ValueError("Dimensions must be positive")
    return length * width
```

# Practical Examples

# Example 1: Temperature Conversion

```
def celsius_to_fahrenheit(celsius):
    """Convert Celsius to Fahrenheit"""
    return (celsius * 9/5) + 32

def fahrenheit_to_celsius(fahrenheit):
    """Convert Fahrenheit to Celsius"""
    return (fahrenheit - 32) * 5/9

# Usage
print(f"32°F = {fahrenheit_to_celsius(32):.1f}°C")
print(f"100°C = {celsius_to_fahrenheit(100):.1f}°F")
```

# Example 2: Password Generator

```
import random
import string
def generate_password(length=12, use_special=True):
    """
    Generate a random password.
    Args:
        length (int): Length of password (default 12)
        use_special (bool): Include special chars (default
    Returns:
        str: Generated password
    """
    chars = string.ascii_letters + string.digits
    if use_special:
        chars += string.punctuation
    return ''.join(random.choice(chars) for _ in range(leng
```

# Conclusion

# Summary

- Functions are essential for modular, reusable code
- Python offers flexible parameter/argument handling
- Proper scoping and return values are crucial
- Advanced features like lambdas and decorators add power
- Following best practices leads to maintainable code

# Next Steps

- Practice writing various functions
- Explore Python’s built-in functions
- Learn about generator functions and closures
- Study function annotations (type hints)
- Experiment with decorators

# Python Modules and Packages

# Understanding Python's Import System

- Python's import system is powerful and flexible
- Based on the concept of a **module search path**
- Key components:
  - `sys.path` - List of directories Python searches for modules
  - `__import__()` - Built-in function that does the actual importing
  - Module cache (`sys.modules`) - Stores already imported modules
- Import process:
  1. Check if module is already in `sys.modules`
  2. Search through `sys.path` for the module
  3. Compile and execute the module (storing in `sys.modules`)

# The Module Search Path

## Viewing sys.path

```
import sys
print(sys.path)
```

Typical search order:

1. Directory containing the input script (or current directory)
2. `PYTHONPATH` environment variable directories
3. Installation-dependent default path
    - Site-packages directory for installed packages

## Modifying the search path

```
import sys
sys.path.append('/path/to/my/modules')
```

# Detailed Import Mechanisms

## Absolute Imports

```
from package.subpackage import module
from package.subpackage.module import function
```

## Relative Imports (within a package)

```
from . import module            # Current package
from .. import module           # Parent package
from ..subpackage import module
```

## Import Best Practices

- Prefer absolute imports for readability
- Use relative imports only within packages
- Avoid circular imports
- Organize imports (standard library, third-party, local)

# Module Execution and Namespace

- When imported, module code is executed **once**
- Creates a module object with its own namespace
- `__name__` attribute:
  - `"__main__"` when run directly
  - Module name when imported

## Module namespace example (module.py)

```
"""Module documentation"""
import sys
def func():
    pass

class MyClass:
    pass

print(f"Namespace: {dir()}")
print(f"Name: {__name__}")
```

# Package Initialization

- `__init__.py` files:
  - Make directories recognizable as packages
  - Can be empty or contain initialization code
  - Executed when package is imported
- Can define `__all__` to control `from package import *`

## Example __init__.py

```
"""Package documentation"""
from .module1 import func1
from .module2 import func2

__all__ = ['func1', 'func2']  # What gets imported with *

# Package-level initialization
print(f"Initializing {__name__}")
```

# Advanced Import Features

## Dynamic Imports

```
# Using __import__()
module_name = "math"
math = __import__(module_name)
print(math.sqrt(16))
# Using importlib
import importlib
math = importlib.import_module("math")
```

## Import Hooks

- Customize import behavior
- Implement `importlib.abc.MetaPathFinder` or `importlib.abc.PathEntryFinder`
- Used by tools like pytest, Django

# Python's Import Cache

- sys.modules acts as a cache
- Already imported modules are stored here
- Can be manipulated (with caution)

## Working with sys.modules

```
import sys
import math
print('math' in sys.modules)  # True
# Force reload
del sys.modules['math']
import math  # Module is reloaded
```

## Warning

Modifying sys.modules can lead to subtle bugs. Use importlib.reload() instead.

# Module Attributes

| Attribute | Purpose |
|---|---|
| `__name__` | Module name |
| `__file__` | Path to module file |
| `__package__` | Package name |
| `__doc__` | Module docstring |
| `__annotations__` | Variable annotations |
| `__dict__` | Module namespace |

## Inspecting a module

```
import math
print(math.__name__)     # 'math'
print(math.__file__)     # Path to math.py
print(math.__doc__)      # Documentation
```

# Common Import Patterns

## Lazy Import Pattern

- Delay imports until needed
- Useful for optional dependencies

## Interface Pattern

- Use `__init__.py` to expose clean API
- Hide implementation details

## Plugin Pattern

- Dynamically discover and load modules
- Using `importlib` or `pkgutil`

# Import Gotchas

## Circular Imports

```
# module_a.py
import module_b
def func_a():
    module_b.func_b()

# module_b.py
import module_a
def func_b():
    module_a.func_a()
```

## Solutions

- Restructure code to remove circularity
- Move imports inside functions
- Use import at module level only

# Python's Module Types

- **Pure Python modules** (.py files)
- **Compiled modules** (.pyc files)
- **Built-in modules** (written in C, compiled into interpreter)
- **Frozen modules** (embedded in custom Python binaries)
- **Namespace packages** (PEP 420, no `__init__.py`)

## Checking module type

```
import math
import mymodule

print(type(math))        # <class 'module'>
print(math.__file__)     # Shows if it's .py or built-in
```

# Working with Packages

## Package Structure

```
my_package/
    __init__.py         # Package initialization
    module1.py          # Regular module
    subpackage/
        __init__.py     # Subpackage initialization
        module2.py
    data/
        file.txt        # Package data files
```

## Accessing Package Data

```
from importlib.resources import files

data = files('my_package.data').joinpath('file.txt')
with data.open() as f:
```

# Modern Python Packaging

## pyproject.toml (PEP 517/518)

```
[build-system]
requires = ["setuptools>=42"]
build-backend = "setuptools.build_meta"

[project]
name = "my_package"
version = "1.0.0"
dependencies = [
    "requests>=2.25.0",
    "numpy>=1.20.0"
]
```

- **setuptools**: Traditional packaging
- **poetry**: Modern dependency management
- **pipenv**: Combines pip and virtualenv

# Package Distribution

- **PyPI** (Python Package Index): Main repository
- **TestPyPI**: For testing package uploads
- Distribution formats:
  - Source distribution (.tar.gz)
  - Wheel (.whl) - Built distribution
- Tools:
  - `twine`: Secure package uploads
  - `build`: Create distribution packages

## Publishing a Package

1. Create `pyproject.toml`
2. Build distributions: `python -m build`
3. Upload: `twine upload dist/*`

# Questions

Any questions?