Unit 1

Advanced Python Concepts and Best Practices

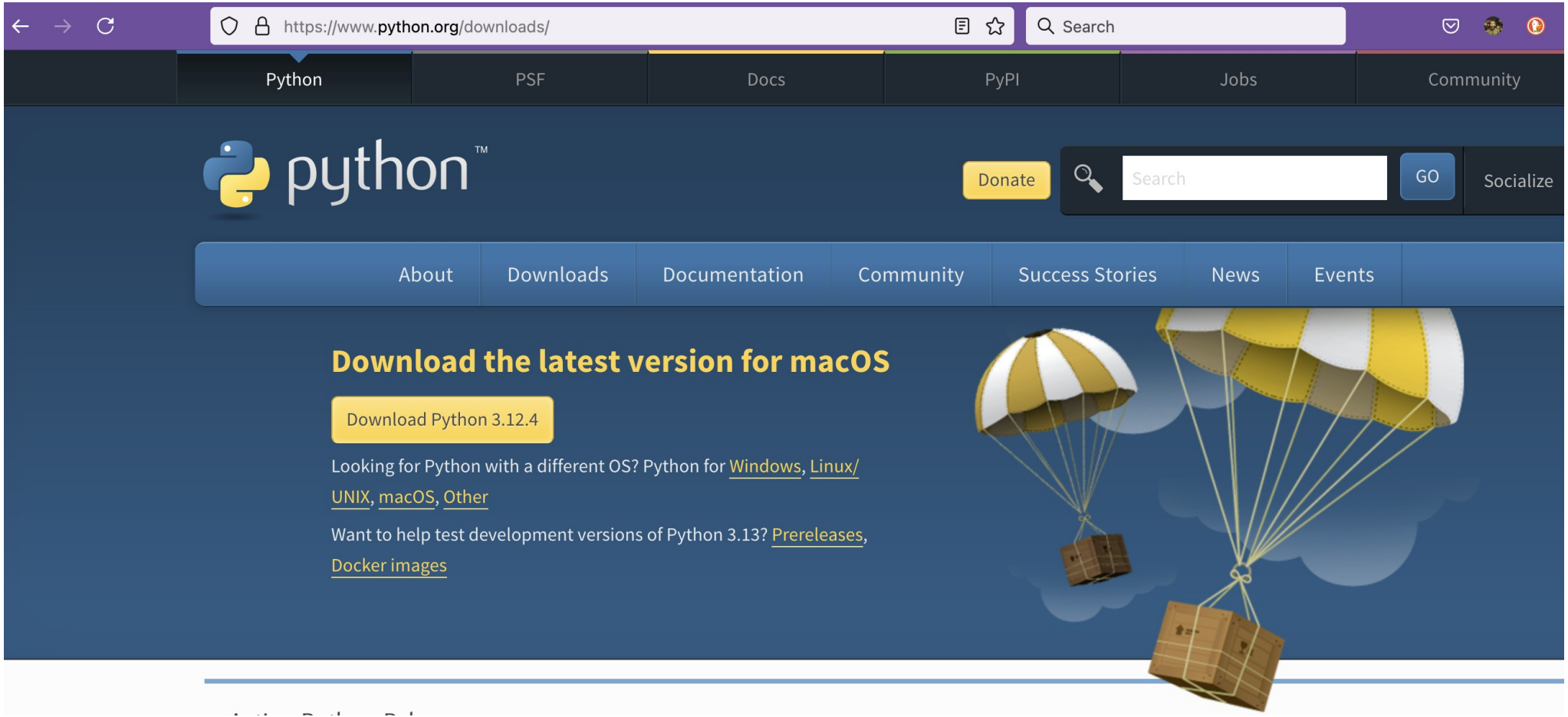
Contents

Contents

Introduction to Python

- Python is a high-level, interpreted programming language known for its readability and simplicity.
- It supports multiple programming paradigms, including procedural, object-oriented, and functional programming.

Downloading Python



Python Releases for Windows

- Latest Python 3 Release - Python 3.12.4

Stable Releases

- Python 3.12.4 - June 6, 2024

  Note that Python 3.12.4 *cannot* be used on Windows 7 or earlier.

  - Download Windows installer (64-bit)
  - Download Windows installer (ARM64)
  - Download Windows embeddable package (64-bit)
  - Download Windows embeddable package (32-bit)
  - Download Windows embeddable package (ARM64)

Pre-releases

- Python 3.13.0b4 - July 17, 2024
  - Download Windows installer (64-bit)
  - Download Windows installer (ARM64)
  - Download Windows embeddable package (64-bit)
  - Download Windows embeddable package (32-bit)
  - Download Windows embeddable package (ARM64)
  - Download Windows installer (32 -bit)
- Python 3.13.0b3 - June 27, 2024

Installing Python

- To get started with Python, you need to install it from python.org.
- After installation, you can check the version using the command:

```
python --version
```

Running Python Code

- You can run Python code in several ways:

- **Interactive Mode:** Open a terminal and type python or python3.
- **Script Mode:** Write your code in a .py file and run it using python filename.py.
- **IDEs:** Use Integrated Development Environments like PyCharm, VSCode, or Jupyter Notebook for a more enhanced coding experience.

Running Python Code: Interactive Mode

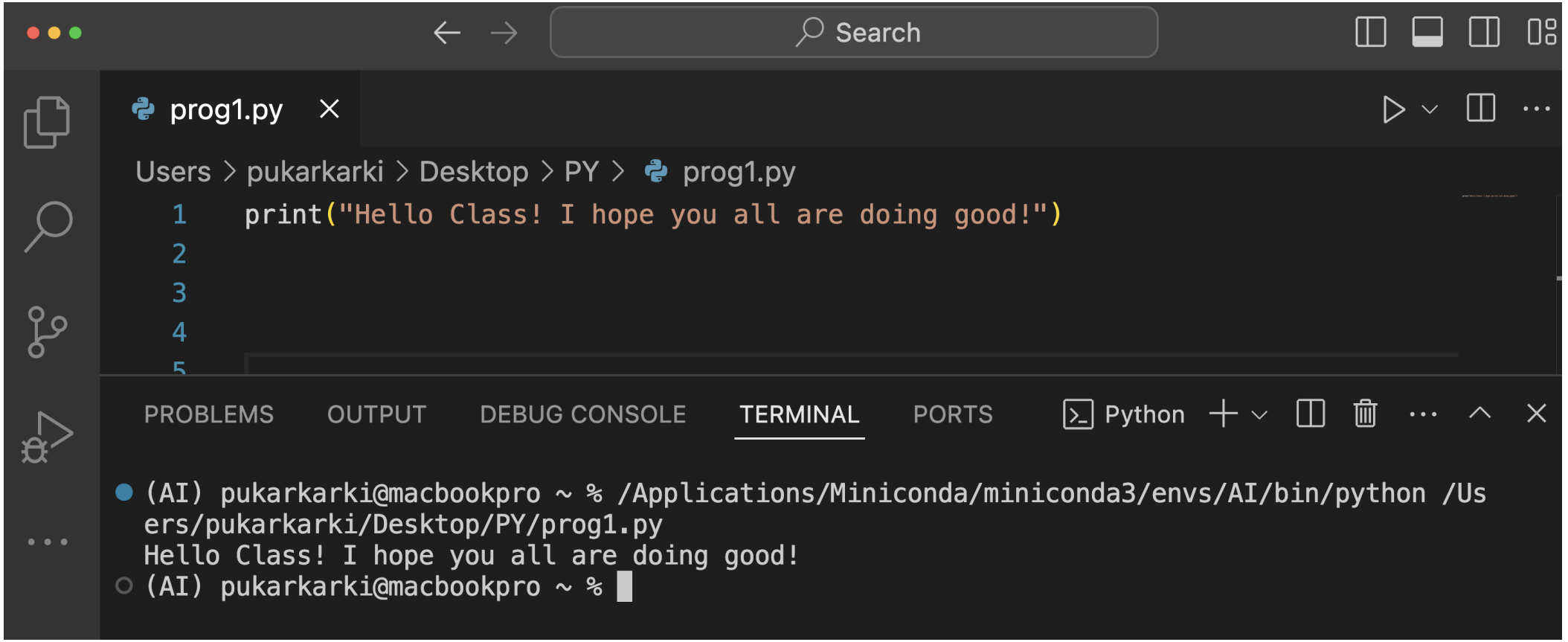
```
pukarkarki@macbookpro ~ % python3
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23) [Clang 13.0.0 (clang-1
300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
>>> 5**5
3125
>>> 5+5
10
>>> 9//2
4
>>> 9/2
4.5
>>> a = "ram"
>>> a*5
'ramramramramram'
```

Running Python Code: Script Mode

```
  UW PICO 5.09                      File: prog1.py                      Modified

print("Hello Class! I hope you all are doing good!")
```

```
pukarkarki@macbookpro PY % python3 prog1.py
Hello Class! I hope you all are doing good!
```

Running Python Code: IDEs



Basic Syntax: Comments

- Comments are used to explain code and are not executed by the interpreter.

prog2.py

Users > pukarkarki > Desktop > PY > prog2.py

```
1  # This is a single-line comment
2
3  """
4  This is a
5  multi-line comment
6  """
7
8  print("I will only be printed")
```

```
pukarkarki@macbookpro PY % python3 prog2.py
I will only be printed
```

Basic Syntax: Variables and Data Types

- Variables are used to store data values. Python has dynamic typing, meaning you don’t have to declare the type of a variable.

```
x = 5           # Integer
y = 3.14        # Float
name = "John"   # String
is_valid = True # Boolean
```

Basic Syntax: Variables and Data Types

- Python supports various arithmetic and logical operations.

```
# Arithmetic operations
sum = 5 + 3
diff = 5 - 3
prod = 5 * 3
quot = 5 / 3
mod = 5 % 3

# Logical operations
and_op = True and False
or_op = True or False
not_op = not True
```

Data Structures: List

- Lists are ordered, mutable collections of items.

prog3.py

Users > pukarkarki > Desktop > PY > prog3.py > ...

```
1    fruits = ["apple", "banana", "cherry"]
2    fruits.append("date")
3    print(fruits)
4    print(fruits[0])
5
```

```
pukarkarki@macbookpro PY % python3 prog3.py
['apple', 'banana', 'cherry', 'date']
apple
```

Data Structures: Tuples

- Tuples are ordered, immutable collections of items.

prog4.py

Users > pukarkarki > Desktop > PY > prog4.py > ...

```
1    coordinates = (10, 20, 30)
2    print(coordinates[0])
3    print(type(coordinates))
4
```

```
pukarkarki@macbookpro PY % python3 prog4.py
10
<class 'tuple'>
```

Data Structures: Sets

- Sets are unordered collections of unique items.

prog5.py

Users > pukarkarki > Desktop > PY > prog5.py > ...

```
1    unique_numbers = {1, 2, 3, 3}
2    print(unique_numbers)
3    unique_numbers.add(4)
4    print(unique_numbers)
5    print(type(unique_numbers))
```

```
pukarkarki@macbookpro PY % python3 prog5.py
{1, 2, 3}
{1, 2, 3, 4}
<class 'set'>
```

Data Structures: Dictionaries

- Dictionaries are unordered collections of key-value pairs.

prog6.py

Users > pukarkarki > Desktop > PY > prog6.py > ...

```python
person = {"name": "John", "age": 30}
print(person["name"])
print(person["age"])
person["address"] = "Dharan"
print(person)
print(type(person))
```

```
pukarkarki@macbookpro PY % python3 prog6.py
John
30
{'name': 'John', 'age': 30, 'address': 'Dharan'}
<class 'dict'>
```

Control Structures: Conditional Statements

- Conditional statements allow you to execute code based on certain conditions.

prog7.py

Users > pukarkarki > Desktop > PY > prog7.py > ...

```
1   email = input("enter your email ")
2   password = input("enrer your password ")
3   if email == "hello@hello.com" and password == "hello123":
4       print("Logged in!")
5   else:
6       print("Invalid Password")
```

```
pukarkarki@macbookpro PY % python3 prog7.py
enter your email hello@hello.com
enrer your password hello123
Logged in!
```

Control Structures: Conditional Statements

- Conditional statements allow you to execute code based on certain conditions.

prog8.py

Users > pukarkarki > Desktop > PY > prog8.py > ...

```
1   yob = int(input("Please enter your Birth Year: "))
2   age = 2081 - yob
3   if age >= 18:
4       print("You are an adult.")
5   elif age >= 13:
6       print("You are a teenager.")
7   else:
8       print("You are a child.")
```

```
pukarkarki@macbookpro PY % python3 prog8.py
Please enter your Birth Year: 2030
You are an adult.
```

Control Structures: Loops

- Loops are used to execute a block of code multiple times.

prog9.py

Users > pukarkarki > Desktop > PY > prog9.py > ...

```
1  for i in range(5):
2      print(i)
3
```

```
pukarkarki@macbookpro PY % python3 prog9.py
0
1
2
3
4
```

Control Structures: Loops

- Loops are used to execute a block of code multiple times.

prog10.py

Users > pukarkarki > Desktop > PY > prog10.py > ...

```
1  count = 0
2  while count < 5:
3      print(count)
4      count += 1
5
6
```

```
pukarkarki@macbookpro PY % python3 prog10.py
0
1
2
3
4
```

Control Structures: Functions

- Functions are reusable blocks of code that perform a specific task.

prog11.py

Users > pukarkarki > Desktop > PY > prog11.py > ...

```
1   def fact(num):
2       f = 1
3       for i in range(1,num+1):
4           f = f*i
5       return f
6
7   n = int(input("Enter a number "))
8   print(f"The factorial of {n} is {fact(n)}")
9
```

```
pukarkarki@macbookpro PY % python3 prog11.py
Enter a number 6
The factorial of 6 is 720
```

Control Structures: Functions

- Functions are reusable blocks of code that perform a specific task.

prog12.py

Users > pukarkarki > Desktop > PY > prog12.py > ...

```python
def average(a,b,c):
    avg = (a+b+c)/3
    return avg


num1 = int(input("Enter num1 "))
num2 = int(input("Enter num2 "))
num3 = int(input("Enter num3 "))
print(f"The average of {num1}, {num2} and {num3} is {average(num1, num2, num3)}")
```

```
pukarkarki@macbookpro PY % python3 prog12.py
Enter num1 3
Enter num2 4
Enter num3 5
The average of 3, 4 and 5 is 4.0
```

Object Oriented Programming: Classes and Objects

- Classes are blueprints for creating objects. Objects are instances of classes.

prog13.py

Users > pukarkarki > Desktop > PY > prog13.py > ...

```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return f"{self.name} - Woof!"

dog1 = Dog("Buddy", 3)
dog2 = Dog("Tommy", 8)
print(dog1.bark())
print(dog2.bark())
```

```
pukarkarki@macbookpro PY % python3 prog13.py
Buddy - Woof!
Tommy - Woof!
```

File I/O – Writing to a file

prog16.py

Users > pukarkarki > Desktop > PY > prog16.py > file

```python
# Writing to a file
flag = True
with open("example.txt", "w") as file:
    file.write("Hello, world!")
    while(flag):
        inp_str = input("Enter data..(Press NO/no to quit)")
        if inp_str == "no" or inp_str == "NO":
            flag = False
        else:
            file.write(inp_str)
```

File I/O – Writing to a file

```
pukarkarki@macbookpro PY % python3 prog16.py
Enter data..(Press NO/no to quit)Where
Enter data..(Press NO/no to quit)The
Enter data..(Press NO/no to quit)Mind
Enter data..(Press NO/no to quit)Is
Enter data..(Press NO/no to quit)Without
Enter data..(Press NO/no to quit)Fear
Enter data..(Press NO/no to quit)NO
```

File I/O – Reading from a File

prog17.py

Users > pukarkarki > Desktop > PY > prog17.py > file

```python
# Reading from a file
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

```
pukarkarki@macbookpro PY % python3 prog17.py
Hello, world!WhereTheMindIsWithoutFear
```

Exception Handling

- Exceptions are errors that occur during the execution of a program.
- Python has built-in support for handling exceptions.

prog18.py

Users > pukarkarki > Desktop > PY > prog18.py > ...

```python
try:
    num1 = int(input("Enter a number "))
    num2 = int(input("Enter a number "))
    result = num1 / num2
    print(f"The result when {num1} is divided by {num2} is {result}")
except ZeroDivisionError:
    print("Cannot divide by zero.")
finally:
    print("This will always execute.")
```

Exception Handling

```
pukarkarki@macbookpro PY % python3 prog18.py
Enter a number 22
Enter a number 5
The result when 22 is divided by 5 is 4.4
This will always execute.
```

```
pukarkarki@macbookpro PY % python3 prog18.py
Enter a number 22
Enter a number 0
Cannot divide by zero.
This will always execute.
```

Libraries and Frameworks

- Python has a rich ecosystem of libraries and frameworks for various tasks:

  - NumPy: Numerical computing
  - Pandas: Data analysis
  - Matplotlib: Data visualization
  - Flask/Django/Streamlit: Web development
  - Scikit-Learn: Machine Learning and Data Mining
  - TensorFlow/PyTorch: Deep Learning

Contents

Collections Module

- Python has a built-in module called collections. It gives you extra data structures that are more powerful than lists and dicts.

1) Counter

2) defaultdict

3) deque

4) namedtuple

Collections Module - Counter

- Counts how many times each item appears.

```
from collections import Counter

nums = [1, 2, 2, 3, 3, 3]
c = Counter(nums)
print(c)
```

✓ 0.0s

```
Counter({3: 3, 2: 2, 1: 1})
```

Collections Module - defaultdict

- Creates default values automatically.

```
from collections import defaultdict

d = defaultdict(int)
d['a'] += 1
print(d)
```

✓ 0.0s

```
defaultdict(<class 'int'>, {'a': 1})
```

Collections Module - deque

- Fast for adding/removing from both sides.

```
from collections import deque

q = deque([1, 2, 3])
q.appendleft(0)
print(q)
```

✓ 0.0s

```
deque([0, 1, 2, 3])
```

Collections Module - namedtuple

- Works like a lightweight object.

```
from collections import namedtuple

Point = namedtuple('Point', 'x y')
p = Point(3, 4)
print(p.x, p.y)
```

✓ 0.0s

```
3 4
```

Iterators

- An iterator lets you loop through items one by one.
- It remembers where it is, so it uses less memory.

```
nums = [10, 20, 30]
it = iter(nums)

print(next(it))
print(next(it))
```

✓ 0.0s

```
10
20
```

Iterators

- A for loop uses an iterator underneath:

```
nums = [10, 20, 30]

for x in nums:
    print(x)
```

- The loop calls iter(nums) and then keeps calling next().

Iterators

- A for loop works using an iterator.
- Under the hood, this is what happens:
  - Python calls iter() on the object.
  - It gets an iterator.
  - It repeatedly calls next() on that iterator.
  - When StopIteration is raised, the loop ends.

Iterators

```
nums = [1, 2, 3]
it = iter(nums)

while True:
    try:
        x = next(it)
        print(x)
    except StopIteration:
        break
```

✓ 0.0s

```
1
2
3
```

Generators

- A generator is a simple way to build an iterator with yield.
- It saves memory because it produces values only when needed.

Generators

```
def squares(n):
    for i in range(n):
        yield i * i

for x in squares(5):
    print(x)
```

✓ 0.0s

```
0
1
4
9
16
```

Decorators

- A decorator adds extra behavior to a function without changing it.

```
def my_decorator(func):
    def wrapper():
        print("before")
        func()
        print("after")
    return wrapper
```

```python
def my_decorator(func):
    def wrapper():
        print("before")
        func()
        print("after")
    return wrapper
```

✓ 0.0s

```python
@my_decorator
def hello():
    print("hello")

hello()
```

✓ 0.0s

```
before
hello
after
```

```python
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print("time:", end - start)
        return result
    return wrapper

@timer
def work():
    for _ in range(1000000):
        pass

work()
```

✓ 0.0s

```
time: 0.015041112899780273
```

Contents

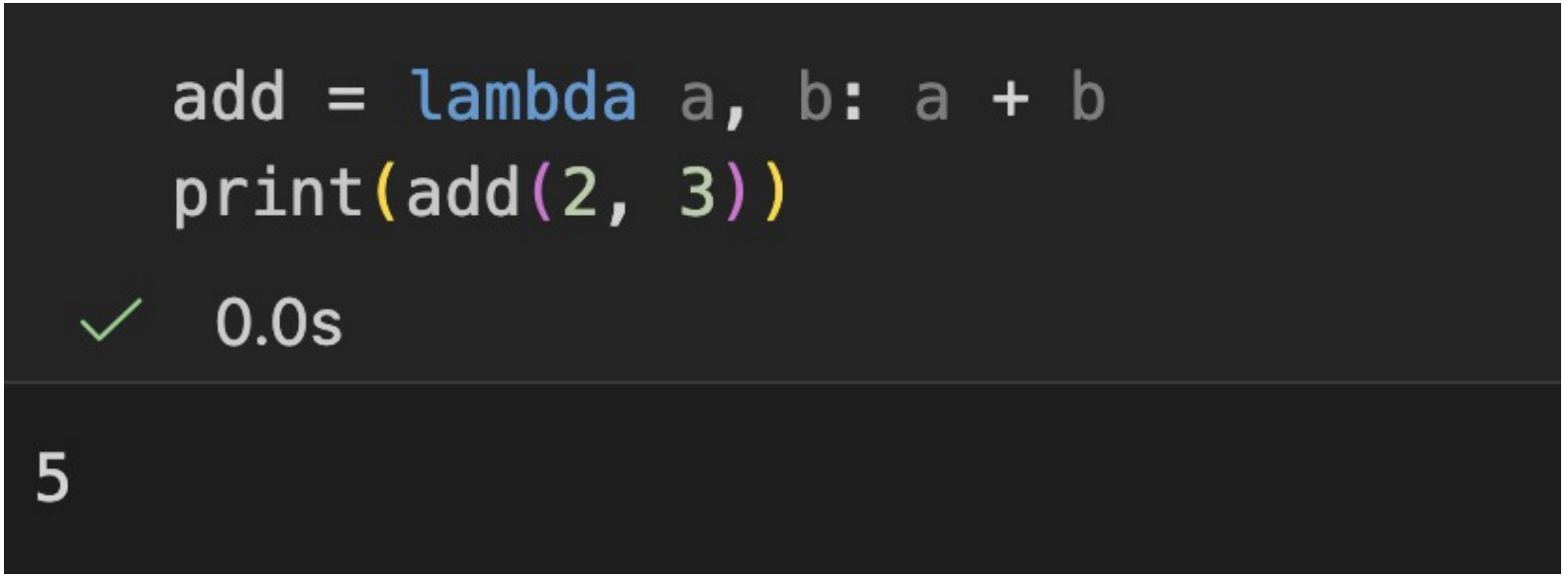
Functions

- A function is a block of code that runs when you call it.
- Functions let you reuse code.
- They make your programs cleaner and easier to manage.

Lambda Expressions

- A lambda is a small, anonymous function.
- You use it when you need a simple function for a short time.

lambda arguments: expression


add = lambda a, b: a + b
print(add(2, 3))
0.0s
5

Common uses - Sorting

```
students = [("Ram", 25), ("Hari", 20)]
students_sorted = sorted(students, key=lambda x: x[1])
students_sorted
```

✓ 0.0s

```
[('Hari', 20), ('Ram', 25)]
```

Common uses - Filtering

```
nums = [1, 2, 3, 4, 5]
evens = list(filter(lambda x: x % 2 == 0, nums))
evens
```

✓ 0.0s

```
[2, 4]
```

Common uses - Mapping

```
nums = range(1,10)
squares = list(map(lambda x: x*x, nums))
squares
```

✓ 0.0s

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

When to use lambda

- Use it only when the function is short and simple.
- If it becomes long, write a normal def function.

Contents

Object-Oriented Programming (OOP) for Data Science

- Class - A class is a blueprint.
- Object - An object is an actual item created from the class.
- Methods - Functions inside a class.
- Attributes - Variables inside a class.

```python
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __str__(self):
        return f"{self.real} + {self.imag}i"

    def __add__(self, other):
        r = self.real + other.real
        i = self.imag + other.imag
        return Complex(r, i)
```

```
a = Complex(3, 4)
b = Complex(1, 2)

print(a)        # 3 + 4i
print(b)        # 1 + 2i

c = a + b
print(c)        # 4 + 6i
```

✓ 0.0s

```
3 + 4i
1 + 2i
4 + 6i
```

Operator Overloading – Artihmetic Operator

Operator	Method	Example
`+`	`__add__`	`p1 + p2`
`-`	`__sub__`	`p1 - p2`
`*`	`__mul__`	`p1 * scalar`
`/`	`__truediv__`	`p1 / scalar`
`//`	`__floordiv__`	`p1 // scalar`
`%`	`__mod__`	`p1 % scalar`
`**`	`__pow__`	`p1 ** 2`

Operator Overloading – Comparison Operator

Operator	Method	Example
`==`	`__eq__`	`p1 == p2`
`!=`	`__ne__`	`p1 != p2`
`<`	`__lt__`	`p1 < p2`
`<=`	`__le__`	`p1 <= p2`
`>`	`__gt__`	`p1 > p2`
`>=`	`__ge__`	`p1 >= p2`

Operator Overloading – Unary Operator

Operator	Method	Example
`-`	`__neg__`	`-p1`
`+`	`__pos__`	`+p1`
`abs()`	`__abs__`	`abs(p1)`

Inheritance

- Inheritance lets a class reuse or extend another class.
- The class being inherited from is called the parent (or base).
- The new class is the child (or derived).

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} is speaking")
```

✓ 0.0s

Inheritance

```
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

    def speak(self):
        print(f"{self.name} is barking")
```

✓ 0.0s

```
d1 = Dog("Lucy", "Shepherd")
d1.speak()
```

✓ 0.0s

```
Lucy is barking
```

Contents

Exception Handling

- Exceptions happen when your code runs into an error.
- Instead of crashing, you can catch them and handle them gracefully.

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
```

✓ 0.0s

```
Cannot divide by zero
```

Basic try-except

Exception Handling

```
try:
    num = int("abc")
    x = 10 / num
except ValueError:
    print("Invalid number")
except ZeroDivisionError:
    print("Division by zero")
```

✓ 0.0s

```
Invalid number
```

Catching multiple exceptions

Exception Handling

```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("Error")
else:
    print("No errors, result =", x)
finally:
    print("This runs always")
```

✓ 0.0s

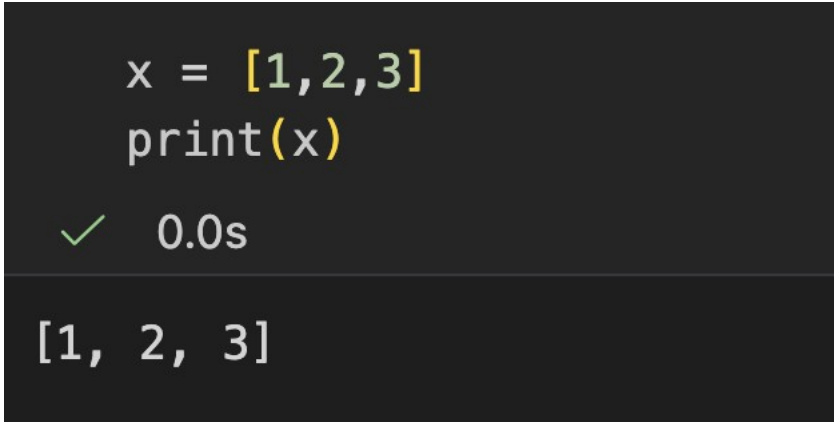
```
No errors, result = 5.0
This runs always
```

try-except-else-finally

```
class DataError(Exception):
    pass

def get_integers():
    aList = []

    while True:
        try:
            s = input("Enter integer (q to quit): ")

            if s == "q":
                break

            try:
                n = int(s)
            except ValueError:
                raise DataError("Invalid integer input")

            aList.append(n)

        except DataError as e:
            print("Caught ",e)

    return aList
```

Debugging

- Debugging helps you find why your code doesn’t work.
- How? Use Print statements - Quick and simple



Debugging - Using pdb

```
import pdb

x = [1, 2, 3, 4]
pdb.set_trace()  # program pauses here

y = x[5]  # this will cause IndexError
```

n → next line (step over)

s → step into function

c → continue until next breakpoint

l → list current code lines

p variable → print value of variable

q → quit debugger

Logging

- Logging is better than print statements for tracking program behavior, especially in production.

```
import logging

logging.basicConfig(level=logging.INFO)
logging.info("Program started")
logging.warning("This is a warning")
logging.error("Something went wrong")
```

✓ 0.0s

```
INFO:root:Program started
WARNING:root:This is a warning
ERROR:root:Something went wrong
```

Logging

- logging.basicConfig() sets up basic logging.

Key parameters:

level: minimum level to log (DEBUG, INFO, WARNING, ERROR, CRITICAL)

format: log message format

filename: write logs to a file instead of console

filemode: "w" overwrite, "a" append

Logging

```python
import logging

logging.basicConfig(
    level=logging.DEBUG,
    format="%(asctime)s %(levelname)s %(message)s",
    filename="app.log",
    filemode="a"
)

logging.debug("debug message")
logging.info("info message")
logging.warning("warning message")
logging.error("error message")
logging.critical("critical message")
```

app.log — Edited

```
2025-12-21 18:50:33,597 DEBUG debug message
2025-12-21 18:50:33,597 INFO info message
2025-12-21 18:50:33,597 WARNING warning message
2025-12-21 18:50:33,597 ERROR error message
2025-12-21 18:50:33,597 CRITICAL critical message
```

Logging

- Logging is better than print statements for tracking program behavior, especially in production.

Contents

Modules and Packages

- Modules are Python files containing functions and variables
- Packages are collections of modules.

prog14.py

Users > pukarkarki > Desktop > PY > prog14.py > ...

```
1   import math
2
3   num = int(input("Number? "))
4   print(f"The square root of {num} is {math.sqrt(num)}")
5
```

```
pukarkarki@macbookpro PY % python3 prog14.py
Number? 625
The square root of 625 is 25.0
pukarkarki@macbookpro PY % python3 prog14.py
Number? 12345
The square root of 12345 is 111.1080555135405
```

Modules and Packages – Creating a Module

- Create a file mymodule.py

mymodule.py

Users > pukarkarki > Desktop > PY > mymodule.py > ...

```
1   def sInterest(p,t,r):
2       si = (p*t*r)/100
3       return si
4
5   def cInterest(p,t,r):
6       ci = p*(((1+r/100))**t - 1)
7       return ci
8
```

Modules and Packages – Creating a Module

- Now we use the module in another file named prog15.py

prog15.py

Users > pukarkarki > Desktop > PY > prog15.py > ...

```
1   import mymodule
2   # from mymodule import sInterest, cInterest
3
4   principal = float(input("Enter Principal "))
5   time = float(input("Enter Time "))
6   rate = float(input("Enter Rate "))
7   print(f"Simple Interest is {mymodule.sInterest(principal, time, rate)}")
8   print(f"Compound Interest is {mymodule.cInterest(principal, time, rate)}")
```

```
pukarkarki@macbookpro PY % python3 prog15.py
Enter Principal 100000
Enter Time 2
Enter Rate 12
Simple Interest is 24000.0
Compound Interest is 25440.00000000002
```