Python Programming

KAUST Academy – King Abdullah University of Science and Technology (KAUST)
Lady Margaret Hall – Oxford University
Presented by: Prof. Naeemullah Khan

Expected Outcomes

By the end of this session, you will be able to:

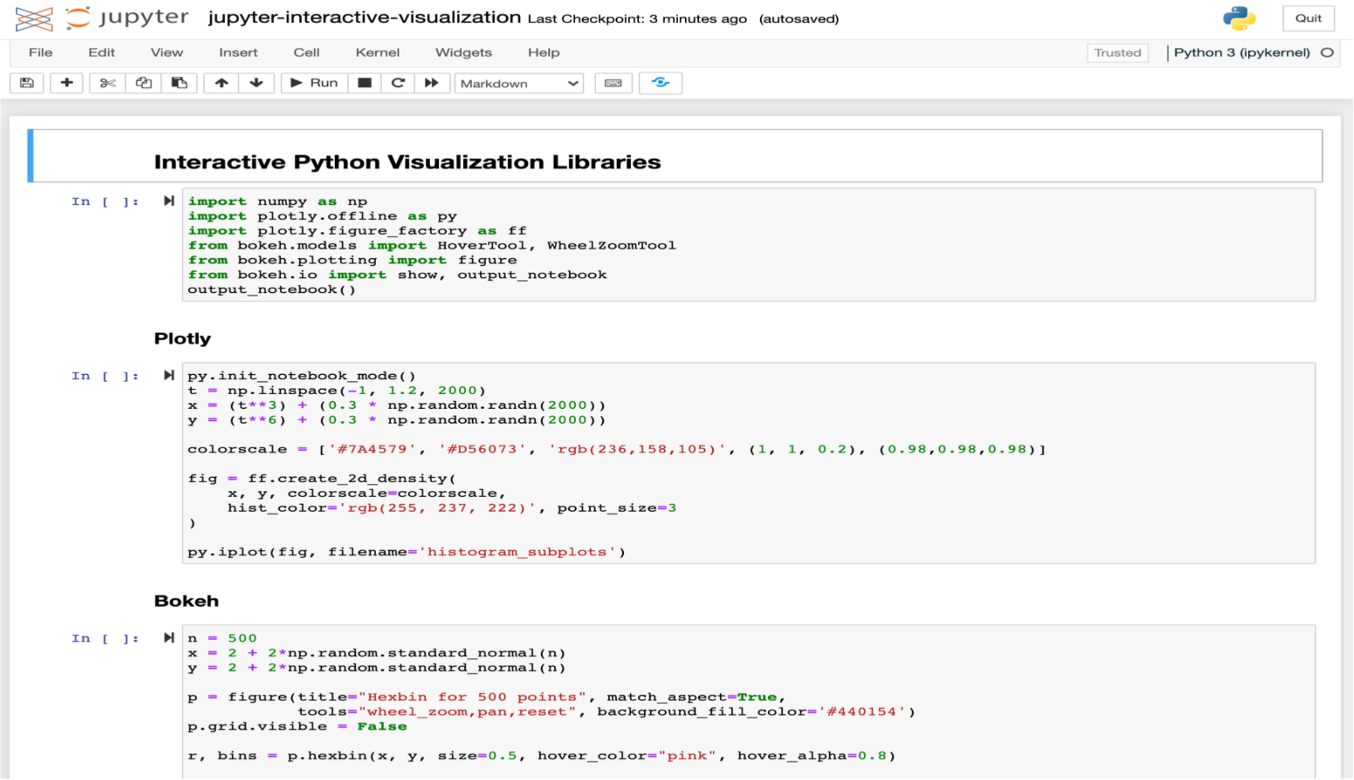
- Understand why Python is important in 2025.
- Explain core syntax, including variables, data types, and collections.
- Write logical structures using conditionals, loops, and functions.
- Understand Object-Oriented Programming (OOP), its components, and its importance.
- Write Python code to solve practical problems.

Why Python Matters for AI in 2025

1. **Rich AI Frameworks:** PyTorch, TensorFlow, JAX, Hugging Face. All are Python-first.

Why Python Matters for AI in 2025

1. **Rich AI Frameworks:** PyTorch, TensorFlow, JAX, Hugging Face. All are Python-first.
2. **Rapid Prototyping:** Clean, readable syntax and interactive Jupyter.



Why Python Matters for AI in 2025

1. **Rich AI Frameworks:** PyTorch, TensorFlow, JAX, Hugging Face. All are Python-first.
2. **Rapid Prototyping:** Clean, readable syntax and interactive Jupyter.
3. **High Performance:** GPU acceleration, Cython/Numba extensions under the hood.

Why Python Matters for AI in 2025

1. **Rich AI Frameworks:** PyTorch, TensorFlow, JAX, Hugging Face. All are Python-first.
2. **Rapid Prototyping:** Clean, readable syntax and interactive Jupyter.
3. **High Performance:** GPU acceleration, Cython/Numba extensions under the hood.
4. **Community & Innovation:** Largest AI community, new research and tutorials arrive in Python first!

Papers With Code

arxiv

Why Python Matters for AI in 2025

1. **Rich AI Frameworks:** PyTorch, TensorFlow, JAX, Hugging Face. All are Python-first.
2. **Rapid Prototyping:** Clean, readable syntax and interactive Jupyter.
3. **High Performance:** GPU acceleration, Cython/Numba extensions under the hood.
4. **Community & Innovation:** Largest AI community, new research and tutorials arrive in Python first!
5. **End-to-End Ecosystem:** From data ingestion (Pandas) to deployment (FastAPI, ONNX Runtime)

Core Language Basics

Numbers, operators, strings and data structures

Data Structures

Name	Type	Description
Integers	int	Whole numbers, such as: **3 300 200**
Floating point	float	Numbers with a decimal point: **2.3 4.6 100.0**
Strings	str	Ordered sequence of characters: **"hello" 'Sammy' "2000" "楽しい"**
Lists	list	Ordered sequence of objects: **[10,"hello",200.3]**
Dictionaries	dict	Unordered Key:Value pairs: **{"mykey" : "value" , "name" : "Frankie"}**
Tuples	tup	Ordered immutable sequence of objects: **(10,"hello",200.3)**
Sets	set	Unordered collection of unique objects: **{"a","b"}**
Booleans	bool	Logical value indicating **True** or **False**

Numbers: Integers and Floats + Logical Operations

```python
# Define integer and float
a = 10
b = 3.0

# Check types
print(type(a), type(b))   # <class 'int'> <class 'float'>

# Arithmetic operations
print(a + b)    # 13.0
print(a - b)    # 7.0
print(a * b)    # 30.0
print(a / b)    # 3.333...
print(a // b)   # 3.0    (floor division)
print(a % b)    # 1.0    (remainder)
print(a ** 2)   # 100    (exponentiation)
```

Numbers: Integers and Floats + Logical Operations

```python
# Comparison operations
print(a > b)     # True
print(a == 10)   # True

# Boolean logic
print((a > b) and (b < 5))    # True
print((a < b) or (a == 10))   # True
```

Numbers: Integers and Floats + Logical Operations

Open: [01-numbers.ipynb]()

Strings

- Strings are sequences of characters, using the syntax of either single quotes or double quotes:
  - **'hello'**
  - **"Hello"**
  - **" I don't do that "**

Strings

- Because strings are **ordered sequences** it means we can using **indexing** and **slicing** to grab sub-sections of the string.
- Indexing notation uses [ ] notation after the string (or variable assigned the string).
- Indexing allows you to grab a single character from the string...

Strings

- These actions use [ ] square brackets and a number index to indicate positions of what you wish to grab.

Character :	**h**	**e**	**l**	**l**	**o**
Index :	**0**	**1**	**2**	**3**	**4**

Strings

- Slicing allows you to grab a subsection of multiple characters, a “slice” of the string.
- This has the following syntax:
  - **[start:stop:step]**
- **start** is a numerical index for the slice start

Strings

```python
# Define a sample string
s = "abcdefgh"

# Basic slice: [start:stop]
print(s[2:5])     # 'cde'  (start=2, stop=5)

# Omitting start or stop
print(s[:4])      # 'abcd' (start defaults to 0)
print(s[4:])      # 'efgh' (stop defaults to end)

# Using a step: [start:stop:step]
print(s[1:7:2])  # 'bdf'  (every 2nd char from index 1 to 6)

# Reverse the string: step of -1
print(s[::-1])   # 'hgfedcba'
```

Strings

Open: 02-strings.ipynb

Data Structures: Lists

Lists

- Lists are ordered sequences that can hold a variety of object types.
- They use [] brackets and commas to separate objects in the list.
  - **[1,2,3,4,5]**
- Lists support indexing and slicing. Lists can be nested and also have a variety of useful methods that can be called off of them.

Lists

```python
# Define a list of integers
lst = [1, 2, 3, 4, 5]

# Indexing (0-based)
print(lst[0])      # 1

# Slicing [start:stop]
print(lst[1:4])    # [2, 3, 4]

# Nesting (a list within a list)
nested = [10, [20, 30], 40]
print(nested[1][0])  # 20

# Common list methods
lst.append(6)      # add to end
print(lst)         # [1, 2, 3, 4, 5, 6]
print(lst.count(3))# 1 occurrence of 3
print(lst.pop())   # removes and returns last element (6)
print(lst)         # [1, 2, 3, 4, 5]
```

Lists

Open: [03-Lists.ipynb]()

Data Structures: Dictionaries

Dictionaries

- Dictionaries are unordered mappings for storing objects. Previously we saw how lists store objects in an ordered sequence, dictionaries use a key-value pairing instead.
- This key-value pair allows users to quickly grab objects without needing to know an index location.
- Dictionaries use curly braces and colons to signify the keys and their associated values.

{'key1':'value1','key2':'value2'}

- So when to choose a list and when to choose a dictionary?

Dictionaries vs Lists

	Dictionary	List
Access	by key	by index
Order	unordered (insertion-ordered)	ordered sequence
Syntax	{key: value, …}	[item1, item2, …]
Use Case	fast lookup / mappings	ordered collection / iteration

Dictionaries

```python
# Example 1: Simple key→value lookup
person_age = {'Alice': 30, 'Bob': 25}
print(person_age['Alice'])      # 30

# Example 2: Dictionary comprehension
squares = {n: n**2 for n in range(1, 6)}
print(squares)                  # {1:1, 2:4, 3:9, 4:16, 5:25}

# Example 3: Using get() for defaults
inventory = {'apples': 5, 'bananas': 2}
inventory['pears'] = inventory.get('pears', 0) + 3
print(inventory)                # {'apples': 5, 'bananas': 2, 'pears': 3}
```

Dictionaries

Open: [04-dictionaries.ipynb]()

Data Structures: Tuples and Sets

Tuples

Tuples are very similar to lists. However they have one key difference - **immutability.**

Once an element is inside a tuple, it can not be reassigned.

Tuples use parenthesis: **(1,2,3)**

Tuples

```python
# Define a tuple of integers
t = (1, 2, 3, 4)

# Indexing (0-based)
print(t[1])        # 2

# Slicing returns a new tuple
print(t[1:3])      # (2, 3)

# Tuple unpacking
a, b, c, d = t
print(a, d)        # 1 4

# Immutability: attempting to change an element raises an error
t[0] = 10  # Will not work
```

Sets

Sets are unordered collections of **unique** elements.

Meaning there can only be one representative of the same object.

Let's see some examples!

Sets

```
# Define a set of integers
s = {1, 2, 3}
print(s)                 # {1, 2, 3}

# Add and remove elements
s.add(4)
s.discard(2)
print(s)                 # {1, 3, 4}

# Membership test
print(3 in s)           # True
print(2 in s)           # False

# Basic set operations
t = {3, 4, 5}
print(s.union(t))        # {1, 3, 4, 5}
print(s.intersection(t)) # {3, 4}
print(s.difference(t))   # {1}
```

Tuples and Sets

Open: [05-Tuples_Sets_Unpacking.ipynb]

Conditional Flow and Functions

Conditional Flow

Often in programming we want to perform cetain actions based on certain conditions, this is called condtional flow or conditional expressions

These conditonal expressions mimic human reasoning or actions, e.g. the python if statement is exactly equivalent to the “if” in the expression, if it rains get an umbrella

If Statement

- Evaluate a **boolean expression** to decide which block runs.
- **Syntax:** if **-->** elif **-->** else
- **Indentation** defines scope of each branch

```
x = int(input("Enter a number: "))
if x > 0:
    print("Positive")
elif x == 0:
    print("Zero")
else:
    print("Negative")
```

For & While Loops

- **for**: loop over items in an iterable (list, range, etc.)
- **while**: repeat as long as a condition holds true
- Use **break** to exit early, **continue** to skip an iteration
- Handle repeated tasks and traversal

```python
# for-loop example
for letter in "dog":
    print(letter)

# while-loop example
count = 0
while count < 3:
    print("Count is", count)
    count += 1
```

Functions

- Use “**def name(params):**” to create reusable block
- **Parameters** input data.
- **Return** sends back results.

```python
def greet(name: str) -> str:
    """Return a greeting message for the given name."""
    return f"Hello, {name}!"

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

If, For, While, and Functions

Open: [06-If_For_While_Functions.ipynb]()

Classes and Objects in Python

OOP, Defining a Class

- Python was built as a procedural language
  - OOP exists and works fine, but feels a bit more "tacked on"

- Declaring a class:

```
class name:
    statements
```

Fields

name = value

- Example:

```
class Point:
    x = 0
    y = 0

# main
p1 = Point()
p1.x = 2
p1.y = -5
```

point.py

```
class Point:
    x = 0
    y = 0
```

- can be declared directly inside class (as shown here) or in constructors (more common)
- Python does not really have encapsulation or private fields
  - relies on caller to "be nice" and not mess with objects' contents

Using a Class

`import` **class**

- client programs must import the classes they use

point_main.py

```
from Point import *

# main
p1 = Point()
p1.x = 7
p1.y = -3
...

# Python objects are dynamic (can add fields any time!)
p1.name = "Tyler Durden"
```

Object Methods

`def` **name**`(self,` **parameter**`,` **...**`,` **parameter**`):`
statements

- `self` *must* be the first parameter to any object method
  - represents the "implicit parameter" (`this` in Java)
- *must* access the object's fields through the `self` reference

```
class Point:
    def translate(self, dx, dy):
        self.x += dx
        self.y += dy
    ...
```

"Implicit" Parameter (`self`)

- Python: `self`, explicit

```
def translate(self, dx, dy):
    self.x += dx
    self.y += dy
```

- Exercise: Write `distance`, `set_location`, and `distance_from_origin` methods.

Exercise Answer

point.py

```
from math import *

class Point:
    x = 0
    y = 0

    def set_location(self, x, y):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return sqrt(self.x * self.x + self.y * self.y)

    def distance(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return sqrt(dx * dx + dy * dy)
```

Calling Methods

- A client can call the methods of an object in two ways:
  - (the value of `self` can be an implicit or explicit parameter)

1) **object**`.`**method**`(`**parameters**`)`

or

2) **Class**`.`**method**`(`**object**`,` **parameters**`)`

- Example:

```
p = Point(3, -4)
p.translate(1, 5)
Point.translate(p, 1, 5)
```

Constructors

```
def __init__(self, parameter, ..., parameter):
    statements
```

- a constructor is a special method with the name `__init__`
- Example:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    ...
```

  - How would we make it possible to construct a `Point()` with no parameters to get (0, 0)?

`toString` and `__str__`

```
def __str__(self):
    return string
```

- equivalent to Java's `toString` (converts object to a string)
- invoked automatically when `str` or `print` is called

Exercise: Write a `__str__` method for `Point` objects that returns strings like `"(3, -14)"`

```
def __str__(self):
    return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Complete Point Class

point.py

```
from math import *

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return sqrt(self.x * self.x + self.y * self.y)

    def distance(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return sqrt(dx * dx + dy * dy)

    def translate(self, dx, dy):
        self.x += dx
        self.y += dy

    def __str__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Operator Overloading

- **operator overloading**: You can define functions so that Python's built-in operators can be used with your class.
  - See also: http://docs.python.org/ref/customization.html

Operator	Class Method
-	`__neg__(self, other)`
+	`__pos__(self, other)`
*	`__mul__(self, other)`
/	`__truediv__(self, other)`

Unary Operators

Operator	Class Method
-	`__neg__(self)`
+	`__pos__(self)`

Operator	Class Method
==	`__eq__(self, other)`
!=	`__ne__(self, other)`
<	`__lt__(self, other)`
>	`__gt__(self, other)`
<=	`__le__(self, other)`
>=	`__ge__(self, other)`

Exercise

- Exercise: **Write a `Fraction` class** to represent rational numbers like 1/2 and -3/8.

- Fractions should always be stored in reduced form; for example, store 4/12 as 1/3 and 6/-9 as -2/3.
  - Hint: A GCD (greatest common divisor) function may help.

- Define `add` and `multiply` methods that accept another `Fraction` as a parameter and modify the existing `Fraction` by adding/multiplying it by that parameter.

- Define `+`, `*`, `==`, and `<` operators.

Generating Exceptions

`raise` **ExceptionType("message")**

- useful when the client uses your object improperly
- types: `ArithmeticError`, `AssertionError`, `IndexError`, `NameError`, `SyntaxError`, `TypeError`, `ValueError`

- Example:

```
class BankAccount:
    ...
    def deposit(self, amount):
        if amount < 0:
            raise ValueError("negative amount")
        ...
```

Inheritance

```
class name(superclass):
        statements
```

- Example:

```
class Point3D(Point):    # Point3D extends Point
    z = 0
    ...
```

- Python also supports *multiple inheritance*

```
class name(superclass, ..., superclass):
        statements
```

(if > 1 superclass has the same field/method, conflicts are resolved in left-to-right order)

Calling Superclass Methods

- methods: **class**.**method**(**object**, **parameters**)
- constructors: **class**.`__init__`(**parameters**)

```
class Point3D(Point):
    z = 0
    def __init__(self, x, y, z):
        Point.__init__(self, x, y)
        self.z = z

    def translate(self, dx, dy, dz):
        Point.translate(self, dx, dy)
        self.z += dz
```

Q&A