Exploring Python's Versatility: Syntax, Object-Oriented Programming, and Machine Learning Applications

Teo Parashkevov

Content

1. Introduction - who am I ?
2. Why Python ?
   a. Advantages over other languages and Ecosystem
   b. Resource management
   c. Speed compared to other languages
3. Syntax and Features
   a. Context Managers
   b. Decorators
4. OOP
   a. Normal / Abstract classes and Interfaces
   b. Encapsulation, Polymorphism, Multiple Inheritance
   c. Dataclasses
5. Machine Learning
   a. Keras / Tensorflow
   b. Scikit-learn

Introduction

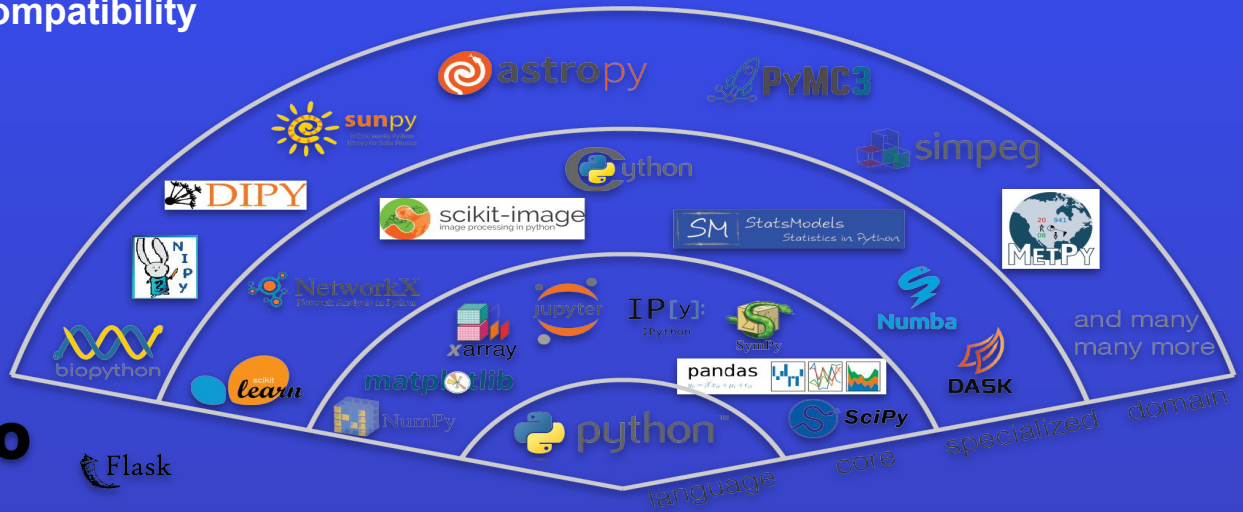
Who is Teo Parashkevov ?

1. Education
   a. German Language School, Ruse, Bulgaria
   b. 东南大学 - Southeast University, Nanjing, China
2. Work experience
   a. System administration
   b. Embedded systems programming
   c. Software engineering - OOP / Functional
   d. Machine Learning and applied Mathematics
3. Current work
   a. @ B EYE Ltd. as Python developer / ML Engineer

Why Python ?

Why Python ?

1. Ease of Learning, Readability and Rapid Development
   a. No {}, -, and other, just : and <TAB>
2. Dynamic Typing and HIgh-level Abstractions for common operations
   a. No explicit declarations. Any variable can be anything - int / str / obj.
3. Cross-platform compatibility
4. Vast ecosystem



django

Flask

Resource management

1. The simplicity of opening / reading / writing / closing files.

The C way of opening and reading the contents of a file

```c
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "r");
    if (file == NULL) {
        perror("Failed to open the file");
        return 1;
    }

    char line[1000];
    while (fgets(line, sizeof(line), file)) {
        printf("%s", line);
    }

    fclose(file);
    return 0;
}

```

The Pythonic way of opening and reading the contents of a file

Python

```python
# Open and read a file line by line in Python
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())  # Remove newline characters

```

We should keep in mind that ...

C

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    int NUMBER, i, s;
    NUMBER = atoi(argv[1]);
    for (s = i = 0; i < NUMBER; ++i) {
        s += 1;
    }
    return 0;
}
```

Python

```python
NUMBER = int(sys.argv[1])
s = 0
for i in range(NUMBER):
    s += 1
```

We should keep in mind that ...

1. **Speed is relevant**
2. 450 mln.
3. Difference of ~40x

```
[teop@teo-surfacepro2 code_1]$ gcc comparison_c.c -o c_exec
[teop@teo-surfacepro2 code_1]$ ls
c_exec  comparison_c.c  comparison_p.py
[teop@teo-surfacepro2 code_1]$ time ./c_exec 450000000

real    0m0,947s
user    0m0,943s
sys     0m0,004s
```

```
[teop@teo-surfacepro2 code_1]$ time python comparison_p.py 450000000

real    0m37,039s
user    0m36,998s
sys     0m0,007s
```

Mojo🔥

Syntax

Context managers

1. Context managers are a powerful feature in Python for resource management and ensuring that resources are properly acquired and released.
2. In C and other languages a similar concept exists but most of the time must be implemented manually.
3. In C and other languages you have to remember to close the file explicitly to release the associated resources, and proper error handling is crucial.

Context managers

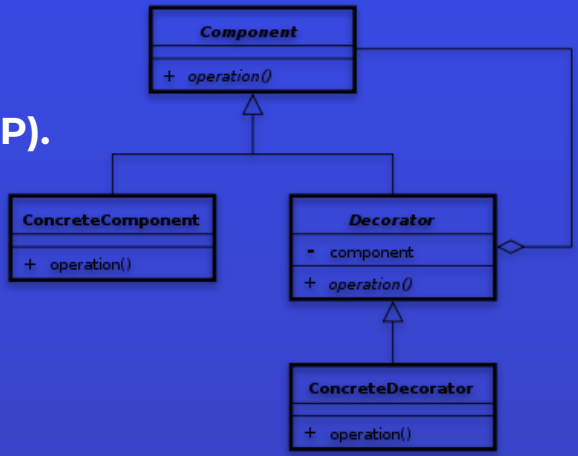
1. **Resource Management**: Simplifies resource acquisition and release.
2. **Automatic Cleanup**: Ensures proper cleanup even during exceptions.
3. **Readability**: Improves code comprehension with clear resource scopes.
4. **Reduced Boilerplate**: Minimizes repetitive resource-handling code.
5. **Safe Practices**: Encourages consistent and safe resource management.
6. **Custom Resources**: Supports custom resource management logic.
7. **Exception Handling**: Integrates with error handling gracefully.
8. **Testing**: Facilitates isolation and mocking in unit tests.
9. **Consistency**: Promotes uniform resource management patterns.
10. **Compatibility**: Works seamlessly with Python's ecosystem and libraries.

Python

```python
# Using a context manager in Python (file handling)
with open("example.txt", "r") as file:
    data = file.read()
# The file is automatically closed when exiting the block

# Using a custom context manager
class MyContextManager:
    def __enter__(self):
        print("Entering the context")
        return self  # This value can be assigned to a variable if needed

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting the context")

with MyContextManager() as cm:
    print("Inside the context")

# Output:
# Entering the context
# Inside the context
# Exiting the context
```

Decorators

1. Python decorators are concise and readable.
2. They separate concerns cleanly.
3. Encourage code reusability.
4. Highly flexible and extensible.
5. Dynamic behavior at runtime.
6. Clear metadata annotations.
7. Rich ecosystem and community.
8. Facilitate Aspect-Oriented Programming (AOP).
9. Enhance code modularity.
10. Align with Python's simplicity.



Decorators Example 1st decorator

https://github.com/theoparashkevov/python-learning-materials/blob/main/Advanced%20Python/syntax/learn-decorators.py

Python

```python
def simple_decorator_4_1(*decorator_args, **decorator_kwargs):
    # there is a function to be 'decorated'

    def decorator(f):
        def wrapper(*function_args, **function_kwargs):
            # Do something
            print('[Decorator (4_1) ] Positional Arguments: ', decorator_args
            print('[Decorator (4_1) ] Keyword Arguments: ', decorator_kwargs

            print('[Decorator (4_1) ] Function: ', f)

            print('[Wrapper (4_1) ] Positional Arguments: ', function_args)
            print('[Wrapper (4_1) ] Keyword Arguments: ', function_kwargs)

            # Run the function
            function_result = f(*function_args, **function_kwargs)

            # Do something on result
            function_result += 0

            return function_result

        return wrapper

    return decorator
```

Decorators Example 2nd decorator

```python
def simple_decorator_4_2(*decorator_args, **decorator_kwargs):
    # there is a function to be 'decorated'

    def decorator(f):
        def wrapper(*function_args, **function_kwargs):
            # Do something
            print('[Decorator (4_2) ] Positional Arguments: ', decorator_args)
            print('[Decorator (4_2)] Keyword Arguments: ', decorator_kwargs)

            print('[Decorator (4_2)] Function: ', f)

            print('[Wrapper (4_2) ] Positional Arguments: ', function_args)
            print('[Wrapper (4_2) ] Keyword Arguments: ', function_kwargs)

            # Run the function
            function_result = f(*function_args, **function_kwargs)

            # Do something on result
            function_result += 0

            return function_result

        return wrapper

    return decorator
```

Decorators
Example
Decorated function

simple yet powerful

```
55 @simple_decorator_4_2('DECORATOR_4_2_ARG_1', 'DECORATOR_4_2_ARG_2')
56 @simple_decorator_4_1('DECORATOR_4_1_ARG_1', 'DECORATOR_4_1_ARG_2')
57 def simple_add_function_4(a: int, b: int) -> int:
58     # simple_decorator(simple_add_function)
59     # wrapper       ->  (a, b)
60     return a + b
```

Decorators Example main

```python
def example_4():
    int_a = 10
    int_b = 20

    print('Using positional arguments')
    result = simple_add_function_4(int_a, int_b)
    print('Final result: ', result, '\n')

    print('Using Keyword arguments')
    result = simple_add_function_4(a=int_a, b=int_b)
    print('Final result: ', result, '\n')
```

Decorators Example output

```
####################################
                  Example 4
####################################
Using positional arguments
[Decorator (4_2) ] Positional Arguments:  ('DECORATOR_4_2_ARG_1', 'DECORATOR_4_2_ARG_2')
[Decorator (4_2)] Keyword Arguments:  {}
[Decorator (4_2)] Function:  <function simple_decorator_4_1.<locals>.decorator.<locals>.wrapper at 0x7f0aff4d4fe0>
[Wrapper (4_2) ] Positional Arguments:  (10, 20)
[Wrapper (4_2) ] Keyword Arguments:  {}
[Decorator (4_1) ] Positional Arguments:  ('DECORATOR_4_1_ARG_1', 'DECORATOR_4_1_ARG_2')
[Decorator (4_1) ] Keyword Arguments:  {}
[Decorator (4_1) ] Function:  <function simple_add_function_4 at 0x7f0aff4d4f40>
[Wrapper (4_1) ] Positional Arguments:  (10, 20)
[Wrapper (4_1) ] Keyword Arguments:  {}
Final result:  30

Using Keyword arguments
[Decorator (4_2) ] Positional Arguments:  ('DECORATOR_4_2_ARG_1', 'DECORATOR_4_2_ARG_2')
[Decorator (4_2)] Keyword Arguments:  {}
[Decorator (4_2)] Function:  <function simple_decorator_4_1.<locals>.decorator.<locals>.wrapper at 0x7f0aff4d4fe0>
[Wrapper (4_2) ] Positional Arguments:  ()
[Wrapper (4_2) ] Keyword Arguments:  {'a': 10, 'b': 20}
[Decorator (4_1) ] Positional Arguments:  ('DECORATOR_4_1_ARG_1', 'DECORATOR_4_1_ARG_2')
[Decorator (4_1) ] Keyword Arguments:  {}
[Decorator (4_1) ] Function:  <function simple_add_function_4 at 0x7f0aff4d4f40>
[Wrapper (4_1) ] Positional Arguments:  ()
[Wrapper (4_1) ] Keyword Arguments:  {'a': 10, 'b': 20}
Final result:  30
```

Object-oriented Programming

Abstract classes / Interfaces

Python does not have built-in support for interfaces like some other statically typed languages (e.g., Java or C#).

Instead, Python relies on duck typing, which means that objects are evaluated based on their behavior rather than their type.

However, you can achieve interface-like behavior using abstract base classes (ABCs) from the abc module or by convention.

Abstract classes Example 1

Python

```python
from abc import ABC, abstractmethod

# Define an abstract class
class Shape(ABC):

    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

# Create a subclass of the abstract class
class Circle(Shape):

    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius * self.radius

    def perimeter(self):
        return 2 * 3.14159 * self.radius
```

Abstract classes Example 2

```python
from abc import ABC, abstractmethod

# Define an abstract base class (interface)
class PaymentGateway(ABC):

    @abstractmethod
    def process_payment(self, amount):
        pass

# Implement the interface
class CreditCardGateway(PaymentGateway):

    def process_payment(self, amount):
        print(f"Processing credit card payment of ${amount}")

class PayPalGateway(PaymentGateway):

    def process_payment(self, amount):
        print(f"Processing PayPal payment of ${amount}")

# Create instances and use them
credit_card = CreditCardGateway()
paypal = PayPalGateway()

credit_card.process_payment(100)
paypal.process_payment(50)

```

Encapsulation

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):
        return self.__balance

account = BankAccount(1000)

```

Encapsulation And properties

Python

```python
class Student:
    def __init__(self, name, age):
        self._name = name  # Protected attribute
        self._age = age

    @property
    def name(self):
        return self._name

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if value >= 0:
            self._age = value

student = Student("Alice", 20)
student.age = 21  # Setter method is used

```

Composition

Python

```python
class Engine:
    def start(self):
        print("Engine started")

class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        self.engine.start()

my_car = Car()
my_car.start()
```

Inheritance

Python

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

class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

my_dog = Dog("Buddy")
```

Multiple Inheritance

```python
# Base class 1
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

# Base class 2
class Flyable:
    def fly(self):
        pass

# Derived class inheriting from both Animal and Flyable
class Bird(Animal, Flyable):
    def speak(self):
        return f"{self.name} says Tweet!"

    def fly(self):
        return f"{self.name} is flying."

# Create an instance of Bird
bird = Bird("Sparrow")

# Call methods from both base classes
print(bird.speak())  # Output: "Sparrow says Tweet!"
print(bird.fly())    # Output: "Sparrow is flying."
```

Polymorphism

Python

```python
class Bird:
    def speak(self):
        return "Bird sound"

class Duck(Bird):
    def speak(self):
        return "Quack!"

def make_bird_speak(bird):
    print(bird.speak())

duck = Duck()
make_bird_speak(duck)  # Output: "Quack!"
```

Dataclasses

In Python, the `dataclass` module is a decorator introduced in Python 3.7 that provides a convenient way to create classes that are primarily used to store data.

It automates the creation of special methods like `__init__()`, `__repr__()`, `__eq__()`, and `__hash__()` that are commonly needed for data classes, reducing boilerplate code and making your code more concise and readable.

Dataclasses

Python

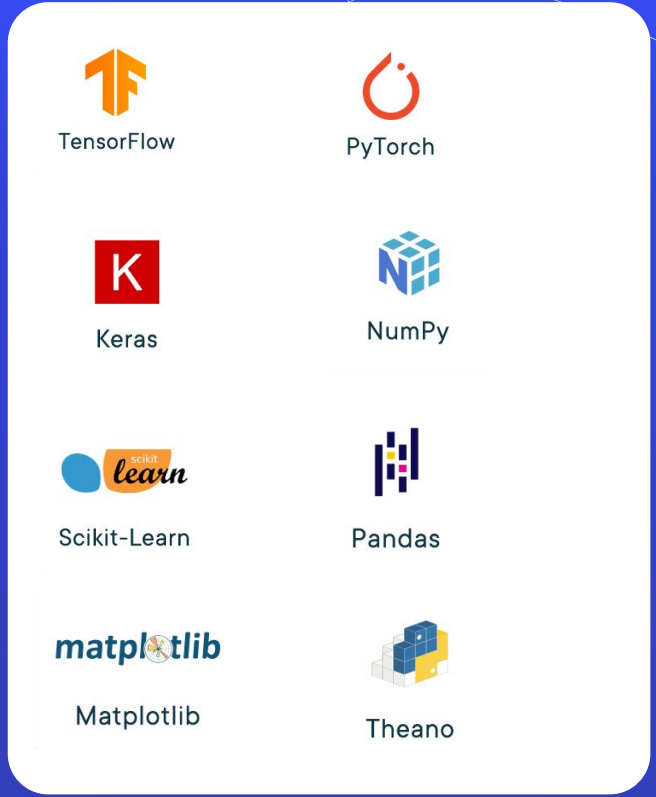
```python
from dataclasses import dataclass, field

@dataclass
class Circle:
    radius: float
    center_x: float = 0.0
    center_y: float = 0.0

# Create a Circle instance with default center coordinates
c = Circle(5.0)
print(c)  # Output: Circle(radius=5.0, center_x=0.0, center_y=0.0)

```

Machine Learning

ML in Python

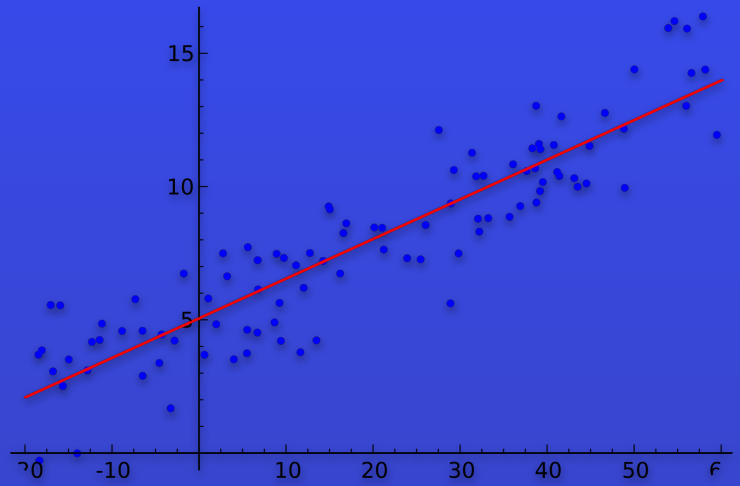
Python is commonly used for machine learning due to its rich ecosystem of libraries like scikit-learn, TensorFlow, and PyTorch, which offer powerful tools for developing and deploying machine learning models.

Additionally, Python's simplicity and readability make it accessible to both beginners and experienced developers, facilitating rapid prototyping and experimentation in the field of machine learning.



Linear Regression

Scikit-Learn



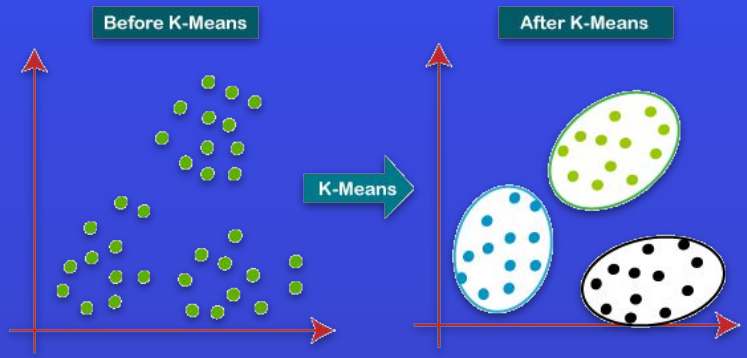
Python

```python
from sklearn.datasets import load_boston
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Load the Boston Housing Prices dataset
boston = load_boston()
X, y = boston.data, boston.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2,
                                                    random_state=42)

# Create and train a linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = model.predict(X_test)

# Evaluate the model using Mean Squared Error
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

Sales Forecasting; Price Optimization; Customer Lifetime Value Prediction; Employee Performance Analysis; Credit Risk Assessment;

K-means Clustering

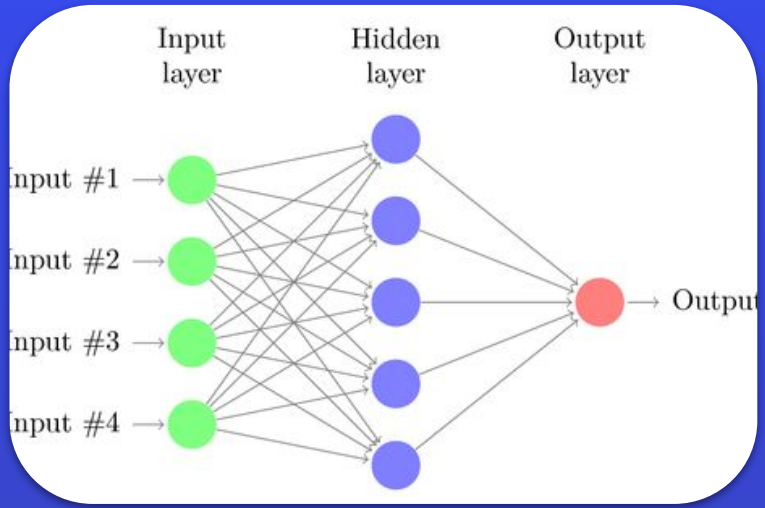
Scikit-Learn




```python
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Generate synthetic data with three clusters
X, _ = make_blobs(n_samples=300,
                  centers=3,
                  random_state=42)

# Create and fit a K-Means clustering model
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)

# Get cluster labels
labels = kmeans.labels_

# Plot the data points and cluster centers
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
plt.scatter(kmeans.cluster_centers_[:, 0],
            kmeans.cluster_centers_[:, 1],
            marker='x', s=200, c='red')
plt.show()
```

Customer Segmentation; Anomaly Detection; Image Compression; Recommendation Systems; Market Basket Analysis

Neural Networks




```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Load the MNIST dataset
mnist = keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Preprocess the data
train_images = train_images / 255.0
test_images = test_images / 255.0

# Build the neural network model
model = keras.Sequential([
    # Flatten the 28x28 input images
    layers.Flatten(input_shape=(28, 28)),
    # Fully connected layer with ReLU activation
    layers.Dense(128, activation='relu'),
    # Dropout layer to prevent overfitting
    layers.Dropout(0.2),
    # Output layer with 10 units for 10 classes (digits)
    layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
model.fit(train_images, train_labels, epochs=5)

# Evaluate the model on the test data
test_loss, test_accuracy = model.evaluate(test_images, test_labels)
print("Test accuracy:", test_accuracy)
```

Thank you

Contacts:


teo-parashkevov


theoparashkevov


teo.parashkevov


Teo Parashkevov


Следете актуалните обяви за **Python**

DEV.BG