ramchandrapadwal

12 PYTHON FEATURES

Every Data Scientist Should Know

1. COMPREHENSIONS

Comprehensions in Python are a useful tool for machine learning and data science tasks as they allow for the creation of complex data structures in a concise and readable manner.

List comprehensions can be used to generate lists of data, such as creating a list of squared values from a range of numbers.

Nested list comprehensions can be used to flatten multidimensional arrays, a common preprocessing task in data science.

```
# list comprehension
_list = [x**2 for x in range(1, 11)]

# nested list comprehension to flatten list
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

flat_list = [num # append to list
             for row in matrix # outer loop
             for num in row] # inner loop

print(_list)
print(flat_list)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Dictionary and set comprehensions are useful for creating dictionaries and sets of data, respectively. For example, dictionary comprehension can be used to create a dictionary of feature names and their corresponding feature importance scores in a machine learning model.

```python
# dictionary comprehension
_dict = {var:var ** 2 for var in range(1, 11) if var % 2 != 0}

# set comprehension
# create a set of squares of numbers from 1 to 10
_set = {x**2 for x in range(1, 11)}

print(_dict)
print(_set)
```

```
{1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
{64, 1, 4, 36, 100, 9, 16, 49, 81, 25}
```

Generator comprehensions are particularly useful for working with large datasets, as they generate values on-the-fly rather than creating a large data structure in memory. This can help to improve performance and reduce memory usage.

```python
# generator comprehension
_gen = (x**2 for x in range(1, 11))

print(list(g for g in _gen))
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

2. ENUMERATE

enumerate is a built-in function that allows for iterating over a sequence (such as a list or tuple) while keeping track of the index of each element.

This can be useful when working with datasets, as it allows for easily accessing and manipulating individual elements while keeping track of their index position.

Here we use **enumerate** to iterate over a list of strings and print out the value if the index is an even number.

```
for idx, value in enumerate(["a", "b", "c", "d"]):
    if idx % 2 == 0:
        print(value)
```

```
a
c
```

ramchandrapadwal

3. ZIP

zip is a built-in function allowing iterating over multiple sequences (such as lists or tuples) in parallel.

Below we use **zip** to iterate over two lists **x** and **y** simultaneously and perform operations on their corresponding elements.

```
x = [1, 2, 3, 4]
y = [5, 6, 7, 8]

# iterate over both arrays simultaneously
for a, b in zip(x, y):
    print(a, b, a + b, a * b)
```

```
1 5 6 5
2 6 8 12
3 7 10 21
4 8 12 32
```

In this case, it prints out the values of each element in **x** and **y**, their sum, and their product.

4. GENERATORS

Generators in Python are a type of iterable that allows for generating a sequence of values on-the-fly, rather than generating all the values at once and storing them in memory.

This makes them useful for working with large datasets that won't fit in memory, as the data is processed in small chunks or batches rather than all at once.

Below we use a generator function to generate the first **n** numbers in the Fibonacci sequence. The **yield** keyword is used to generate each value in the sequence one at a time, rather than generating the entire sequence at once.

```
def fib_gen(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b


res = fib_gen(10)
print(list(r for r in res))
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

5. LAMBDA FUNCTIONS

lambda is a keyword used to create anonymous functions, which are functions that do not have a name and can be defined in a single line of code.

They are useful for defining custom functions on-the-fly for feature engineering, data preprocessing, or model evaluation.

Below we use **lambda** to create a simple function for filtering even numbers from a list of numbers.

```
numbers = range(10)

even_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even_numbers)
```

```
[0, 2, 4, 6, 8]
```

Here’s another code snippet for using lambda functions with Pandas

```
import pandas as pd

data = {
    "sales_person": ["Alice", "Bob", "Charlie", "David"],
    "sale_amount": [100, 200, 300, 400],
}
df = pd.DataFrame(data)

threshold = 250
df["above_threshold"] = df["sale_amount"].apply(
    lambda x: True if x >= threshold else False
)
df
```

	sales_person	sale_amount	above_threshold
0	Alice	100	False
1	Bob	200	False
2	Charlie	300	True
3	David	400	True

6. MAP, FILTER, REDUCE

The functions **map**, **filter**, and **reduce** are three built-in functions used for manipulating and transforming data.

map is used to apply a function to each element of an iterable, **filter** is used to select elements from an iterable based on a condition, and **reduce** is used to apply a function to pairs of elements in an iterable to produce a single result.

Below we use all of them in a single pipeline, calculating the sum of squares of even numbers.

```python
numbers = range(10)

# Use map(), filter(), and reduce() to preprocess and aggregate the list of numbers
even_numbers = filter(lambda x: x % 2 == 0, numbers)
squares = map(lambda x: x**2, even_numbers)
sum_of_squares = reduce(lambda x, y: x + y, squares)


print(f"Sum of the squares of even numbers: {sum_of_squares}")
```

```
Sum of the squares of even numbers: 120
```

7. ANY AND ALL

any and **all** are built-in functions that allow for checking if any or all elements in an iterable meet a certain condition.

any and **all** can be useful for checking if certain conditions are met across a dataset or a subset of a dataset. For example, they can be used to check if any values in a column are missing or if all values in a column are within a certain range.

Below is a simple example of checking for the presence of any even values and all odd values.

```
data = [1, 3, 5, 7]
print(any(x % 2 == 0 for x in data))
print(all(x % 2 == 1 for x in data))
```

```
False
True
```

8. NEXT

next is used to retrieve the next item from an iterator. An iterator is an object that can be iterated (looped) upon, such as a list, tuple, set, or dictionary.

next is commonly used in data science for iterating through an iterator or generator object. It allows the user to retrieve the next item from the iterable and can be useful for handling large datasets or streaming data.

Below, we define a generator **random_numbers()** that yields random numbers between 0 and 1. We then use the next() function to find the first number in the generator greater than 0.9

```python
import random

def random_numbers():
    while True:
        yield random.random()

# Use next() to find the first number greater than 0.9
num = next(x for x in random_numbers() if x > 0.9)

print(f"First number greater than 0.9: {num}")
```

```
First number greater than 0.9: 0.976067069456332
```

9. DEFAULTDICT

defaultdict is a subclass of the built-in **dict** class that allows for providing a default value for missing keys.

defaultdict can be useful for handling missing or incomplete data, such as when working with sparse matrices or feature vectors. It can also be used for counting the frequency of categorical variables.

An example is counting the frequency of items in a list. **int** is used as the default factory for the **defaultdict**, which initializes missing keys to 0.

```
from collections import defaultdict

count = defaultdict(int)
for item in ['a', 'b', 'a', 'c', 'b', 'a']:
    count[item] += 1

count
```

```
defaultdict(int, {'a': 3, 'b': 2, 'c': 1})
```

10. PARTIAL

partial is a function in the **functools** module that allows for creating a new function from an existing function with some of its arguments pre-filled.

partial can be useful for creating custom functions or data transformations with specific parameters or arguments pre-filled. This can help to reduce the amount of boilerplate code needed when defining and calling functions.

Here we use **partial** to create a new function **increment** from the existing **add** function with one of its arguments fixed to the value 1.

Calling **increment(1)** is essentially calling **add(1, 1)**

```
from functools import partial

def add(x, y):
    return x + y

increment = partial(add, 1)
increment(1)
```

```
2
```

11. LRU_CACHE

lru_cache is a decorator function in the **functools** module that allows for caching the results of functions with a limited-size cache.

lru_cache can be useful for optimizing computationally expensive functions or model training procedures that may be called with the same arguments multiple times.

Caching can help to speed up the execution of the function and reduce the overall computational cost.

Here's an example of efficiently computing Fibonacci numbers with a cache (known as memoization in computer science)

```python
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

fibonacci(1e3)
```

```
4.346655768693743e+208
```

12. DATACLASSES

The **@dataclass** decorator automatically generates several special methods for a class, such as **__init__**, **__repr__**, and **__eq__**, based on the defined attributes.

This can help to reduce the amount of boilerplate code needed when defining classes. **dataclass** objects can represent data points, feature vectors, or model parameters, among other things.

In this example, **dataclass** is used to define a simple class **Person** with three attributes: **name**, **age**, and **city**.

```
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int
    city: str

p = Person("Alice", 30, "New York")
print(p)
```

```
Person(name='Alice', age=30, city='New York')
```