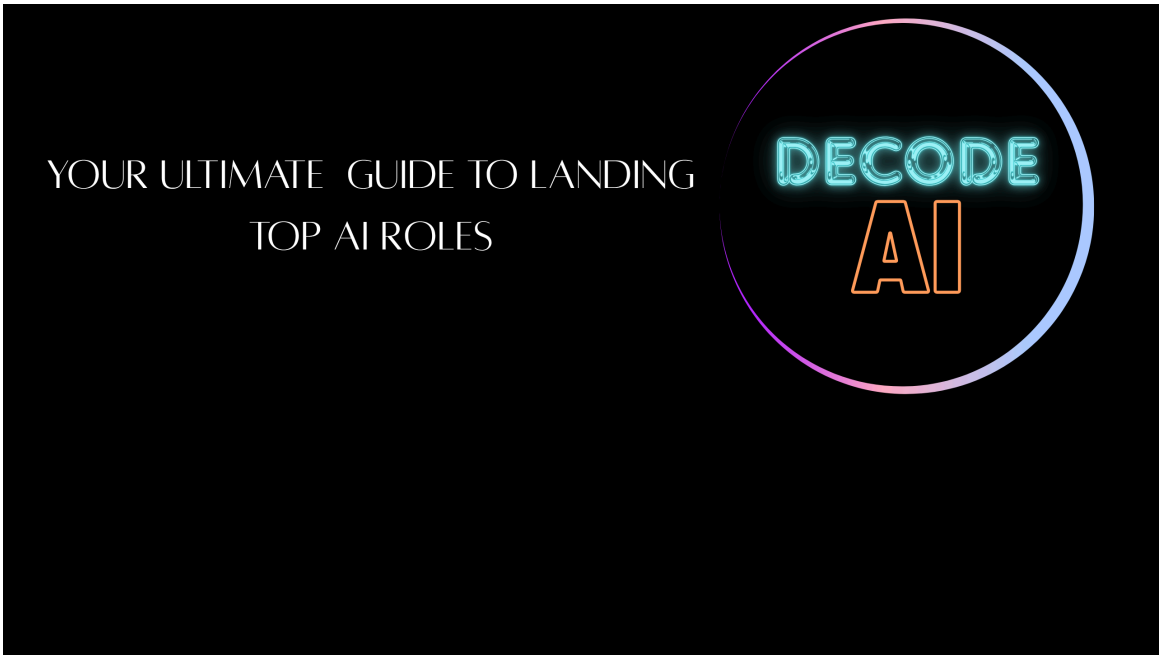

1. Introduction to Functions

A **function** is a reusable block of code that performs a specific task.

Functions help make your code:

- Modular
- Reusable
- Easier to read and maintain

Why Use Functions?

- To avoid repetition (DRY: Don't Repeat Yourself)
- To break complex problems into smaller, manageable chunks
- To make code reusable and organized

Function Syntax in Python

```
def function_name(parameters):
    """
    Optional docstring describing what the function does.
    """
    # Block of code
    return result
```

Important Points

1. We begin with def then a space followed by the name of the function. Try to keep names relevant and simple as possible.
2. Also be careful with names, you wouldn't want to call a function the same name as a built-in function in Python (such as len).
3. Arguments separated by a comma within a pair of parenthesis which acts as input to the defined function, reference them and the function definition with a colon.
4. Indent to begin the code inside the defined functions properly.
5. The doc-string where you write the basic description of the function. Using iPython and iPython Notebooks, you'll be able to read these doc-strings by pressing Shift+Tab after a function name. It is not mandatory to include docstrings with simple functions, but it is a good practice to put them as this will help the programmers to easily understand the code you write.

Problem: You need to write a Industry grade function which takes a string as input, should preprocess it by kepping only alphanumeric english and hindi characters.

In [ ]:
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import re
import logging

# Configure logging
logging.basicConfig(level=logging.INFO, format='[%(levelname)s] %(message)s')


def sanitize_input(user_input: str) -> str:
    """
    Cleans user input by removing unwanted special characters
    while preserving basic punctuation and Unicode characters.

    Args:
        user_input (str): Raw user input string

    Returns:
        str: Sanitized string
    """
    # Step 1: Keep only English, Hindi characters, and spaces
    pattern = r"[^A-Za-z0-9\u0900-\u097F]"
    cleaned = re.sub(pattern, ' ', user_input, flags=re.UNICODE)

    # Step 2: Replace multiple spaces with a single space
    normalized = re.sub(r'\s+', ' ', cleaned)

    return normalized.strip()
```

In [18]:
```
sanitize_input("नमस्कार !! 🙏 Thank you for learning with us! -> Team 🚀 Decode-AI; ")
```

Out[18]:
```
'नमस्कार Thank you for learning with us Team Decode AI'
```

Function vs Method

Hands-on Time

In [19]:
```
#A simple Print Hello function
def say_hello():
    print('hello')
```

In [20]:
```
say_hello()
```

```
hello
```

In [24]:
```
# A simple greeting function
def greeting(name):
    print('Hello %s' %name)
```

In [26]:
```
greeting("Sanjeev")
```

```
Hello Sanjeev
```

In [27]:
```
# Using return in a function
def add_num(num1,num2):
    return num1+num2
```

In [28]:
```
add_num(4,5)
```

Out[28]:
```
9
```

In [29]:
```
# Can also save as variable due to return
result = add_num(4,5)
```

In [30]:
```
result
```

Out[30]:
```
9
```

In [32]:
```python
# Program to check whether a number is prime or not
def is_prime(num):
    '''
    Naive method of checking for primes.
    '''
    for n in range(2,num):
        if num % n == 0:
            print('not prime')
            break
    else: # If never mod zero, then prime
        print('prime')
```

In [33]:
```python
is_prime(12)
```

```
not prime
```

1 Built-in Functions

These are functions that come pre-installed with Python.

📌 Examples:

`print()`, `len()`, `type()`, `max()`, `min()`, `sorted()`

2 User-defined Functions

In [36]:
```python
# These are functions created using the def keyword.
def greet(name):
    return f"Hello, {name}"

greet("sanjeev")
```

Out[36]:
```
'Hello, sanjeev'
```

3 Anonymous Functions (Lambda)

In [38]:
```python
# These are one-liner functions without a name, created using the lambda keyword.
greet_lambda = lambda name: f"Hello, {name}"
greet_lambda("sanjeev")
```

Out[38]:
```
'Hello, sanjeev'
```

4 Recursive Functions

In [40]:
```python
# A function that calls itself to solve smaller sub-problems of a larger problem.
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

factorial(10)
```

Out[40]:
```
3628800
```

In [ ]:

5 map() function

The map() is a function that takes in two arguments:

1. A function
2. A sequence iterable.

In the form: map(function, sequence)

The first argument is the name of a function and the second a sequence (e.g. a list). map() applies the function to all the elements of the sequence. It returns a new list with the elements changed by the function.

In [84]:
```
# map example
def fahrenheit(T):
    return ((float(9)/5)*T + 32)
def celsius(T):
    return (float(5)/9)*(T-32)

temp = [0, 22.5, 40,100]
```

In [88]:
```
F_temps =list(map(fahrenheit, temp))
F_temps
```

Out[88]:
```
[32.0, 72.5, 104.0, 212.0]
```

In [89]:
```
# Convert back
list(map(celsius, F_temps))
```

Out[89]:
```
[0.0, 22.5, 40.0, 100.0]
```

In [92]:
```
# map() with lambda function
a = [1,2,3,4]
b = [5,6,7,8]
c = [9,10,11,12]

sum_lambda = lambda x,y:x+y

ans = []
for i in range(len(a)):
    ans.append(sum_lambda(a[i],b[i]))

ans
```

Out[92]:
```
[6, 8, 10, 12]
```

In [93]:
```
list(map(lambda x,y:x+y,a,b))
```

Out[93]:
```
[6, 8, 10, 12]
```

6 reduce() function

The function reduce(function, sequence) continually applies the function to the sequence. It then returns a single value.

In [94]:
```
# reduce() examples

from functools import reduce
lst =[47,11,42,13]
reduce(lambda x,y: x+y,lst)
```

Out[94]:
```
113
```

7 filter() function

The function filter(function, list) offers a convenient way to filter out all the elements of an iterable, for which the function returns "True".

The function needs to return a Boolean value (either True or False). This function will be applied to every element of the iterable. Only if the function returns "True" will the element of the iterable be included in the result.

In [96]:
```
#First let's make a function
def even_check(num):
    if num%2 ==0:
        return True
```

In [98]:
```
lst =list(range(20))
lst
```

Out[98]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

In [99]:
```
list(filter(even_check,lst))
```

Out[99]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

filter() is more commonly used with lambda functions, this because we usually use filter for a quick job where we don't want to write an entire function.

In [100…
```
list(filter(lambda x: x%2==0,lst))
```

Out[100… [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [ ]:

PART 2

8️⃣ Generators and Iterators

In Python, iterators and generators are tools for iterating over data, but they work differently and serve different purposes.

Iterator Introduction

An iterator is an object which implements the iterator protocol, i.e., it must have:

- a `__iter__()` method that returns the iterator object itself.
- a `__next__()` method that returns the next item or raises StopIteration when done.
- `__iter__()` and `__next__()` — the two special (magic) methods -- the backbone of Python's iterator protocol.
- Key Points:

Term	Definition
Iterable	An object that implements the `__iter__()` method and returns an iterator.
Iterator	An object that implements **both** `__iter__()` and `__next__()` methods.

- List is an iterable not iterator

In [122…
```
# Custom Iterator Examples # iterator which iterates from 1 to n
```

In [1]:
```
class CountUpTo:
    def __init__(self, max):
        self.max = max
        self.current = 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.max:
            raise StopIteration
        else:
            val = self.current
            self.current += 1
            return val
```

In [11]:
```
counter = CountUpTo(3)
counter
```

Out[11]: <__main__.CountUpTo at 0x23fb35ed450>

In [3]:
```
next(counter) # Calls counter.__next__()
```

Out[3]: 1

In [4]:
```
for num in counter:
    print(num)  # Output: 1, 2, 3
```

```
2
3
```

In [5]:
```
# Step-by-step example using iterator
```

In [6]:
```
# Step 1: A list (iterable)
numbers = [1, 2, 3]
print(type(numbers))
```

```
<class 'list'>
```

In [7]:
```
# Step 2: Create an iterator from iterable
it = iter(numbers)  # same as numbers.__iter__()
print(type(it))
```

```
<class 'list_iterator'>
```

In [8]:
```
# Step 3: Use next() to get elements one by one
print(next(it))  # 1
print(next(it))  # 2
print(next(it))  # 3
# print(next(it))  # Raises StopIteration
```

```
1
2
3
```

`next()` and `iter()` built-in functions

In [9]:
```
def simple_gen():
    for x in range(3):
        yield x
```

In [10]:
```
# Assign simple_gen
g = simple_gen()
```

In [63]:
```
# show
g
```

Out[63]:
```
<generator object simple_gen at 0x000002267C91FB80>
```

In [12]:
```
# The next function allows us to access the next element in a sequence.
print(next(g))
```

```
0
```

In [13]:
```
print(next(g))
```

```
1
```

In [14]:
```
print(next(g))
```

```
2
```

In [15]:
```
print(next(g))
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[15], line 1
----> 1 print(next(g))

StopIteration:
```

- After yielding all the values next() caused a StopIteration error. What this error informs us that all the values have been yielded.

In [16]:
```
# why don't we get this error while using a for loop? The "for loop" automatically catches this error
s = 'hello'

#Iterate over string
for let in s:
    print(let)
```

```
h
e
l
l
o
```

- But that doesn't mean the string itself is an iterator! We can check this with the next() function

In [17]:
```
next(s)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[17], line 1
----> 1 next(s)

TypeError: 'str' object is not an iterator
```

- This means that a string object supports iteration, but we can not directly iterate over it as we could with a generator function. The iter() function allows us to do just that!

In [18]:
```
s_iter = iter(s)
```

In [19]:
```
print(type(s_iter))
```

```
<class 'str_ascii_iterator'>
```

In [20]:
```
next(s_iter)
```

Out[20]:
```
'h'
```

In [21]:
```
next(s_iter)
```

Out[21]:
```
'e'
```

Generator Introduction

- A generator is a simpler way to write an iterator using functions and the yield keyword. Each time yield is called, the function's state is saved, and it resumes from there on the next call.
- In Python, Generator function allow us to write a function that can send back a value and then later resume to pick up where it was left. It also allows us to generate a sequence of values over time. The main difference in syntax will be the use of a yield statement.
- In most aspects, a generator function will appear very similar to a normal function.
- The main difference is when a generator function is called and compiled they become an object that supports an iteration protocol.
- That means when they are called they don't actually return a value and then exit, the generator functions will automatically suspend and resume their execution and state around the last point of value generation.

In [ ]:
```
# Generator function for the cube of numbers (power of 3)
```

In [22]:
```
# Normal Implementation
def gencubes(n):
    ans = []
    for num in range(n):
        ans.append(num**3)
    return ans
```

In [23]:
```
# Generator Implementation
def gencubes(n):
    for num in range(n):
        yield num**3
```

In [24]:
```
gencubes(10)
```

Out[24]:
```
<generator object gencubes at 0x0000023FB3B2E5A0>
```

In [25]:
```
for x in gencubes(10):
    print(x)
```

```
0
1
8
27
64
125
216
343
512
729
```

In [26]:
```
# Generator function for the fibonnaci sequence up to n => [1,1,2,3,5,8,....]
```

In [27]:
```
# normal implementation
def fibon(n):
    a = 1
    b = 1
    output = []

    for i in range(n):
        output.append(a)
        a,b = b,a+b

    return output
```

In [28]:
```
fibon(10)
```

Out[28]:
```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

In [29]:
```
# generator Implementation
def genfibon(n):
    '''
    Generate a fibonacci sequence up to n
    '''
    a = 1
    b = 1
    for i in range(n):
        yield a
        a,b = b,a+b
```

In [30]:
```
genfibon(10)
```

Out[30]:
```
<generator object genfibon at 0x0000023FB37E2F80>
```

In [31]:
```
for num in genfibon(10):
    print(num)
```

```
1
1
2
3
5
8
13
21
34
55
```

In [ ]:

🙏 Thank you for learning with us!

— Team 🚀 Decode-AI