Python Fundamentals (Part2)

Concepts : Conditional Statements, Loops & Functions

Conditional Statements

Conditional statements let our program **decide what to do based on conditions** (i.e. true or false expressions).

There are 3 main conditional statements in Python:

1. `if` - used to test a condition. If it's **True**, the indented block runs.
2. `else` - else block runs if all above conditions are false.
3. `elif` - ("else if") used when we have **multiple conditions** to check.

Let's practice conditional statements with the help of following examples.

```
# Example 1
age = int(input("enter age: "))

if age >= 18:            # if true/false, entire block of code is executed
    print("you can vote")
    print("you can drive")

else:
    print("you can't vote")
    print("you can't drive")
```

```
# Example 2 - Traffic Lights
color = input("enter color: ")

if color == "red":
    print("Stop")
elif color == "yellow":
    print("Look")
elif color == "green":
    print("Go")
else:
    print("wrong color")
```

Note - We can have standalone & multiple `if` statements but `elif` & `else` can't be used without an `if` statement preceding them.

Apart from simple conditions we can also check for multiple conditions using Logical operators like `and` , `or` and `not` in expressions.

```
# Example 3 - Logical operators in expressions
age = int(input("enter age: "))

if age < 13:
    print("child")
elif (age >= 13 and age < 18):
    print("teenager")
else:
    print("adult")
```

Nesting in Conditionals

Nesting means **placing one block of code inside another**. In the context of conditional statements, it means putting one `if` (or `if-elif-else`) inside another. This helps when you need to make a decision based on another decision.

```
# Login System

username = input("enter username: ")
password = input("enter password: ")

if (username == "admin" and password == "pass"):
   print("log in successful!")
else:
 if username != "admin":          # NESTING
        print("wrong user name, try again.")
    else:
        print("wrong password, try again.")
```

A login system that checks for correct/ incorrect credentials

Ternary Statements

Ternary statements (or conditional expressions) are just another compact way to writing our `if-else` statements in one-line.

Ternary statement **Syntax**:

```
value_if_true if condition else value_if_false
```

One line decided between 2 values based on condition.

Let's see an example:

```
age = 18
status = "Adult" if age >= 18 else "Not Adult"
print(status)
```

> Note - Everything that we write with ternary statements can be done with normal if-else statements. Ternary statements are generally only used for simple conditions, not recommended for nested or complex conditions.

Match Case Statements

In Python, the `match` / `case` statements are an alternative to long chain of `If..elif..else` statements. They were introduced in Python 3.10 & we generally don't see them much in production code.

Match case **Syntax**:

```
match variable:
   case pattern1:
        # code to run if pattern1 matches
   case pattern2:
       # code to run if pattern2 matches
   case _:
       # default case (like 'else')
```

1. `match` - what you're comparing
2. `case` - the value or pattern to match against
3. `_` - wildcard (matches anything, like "default")

Let's see an example:

```
color = input("enter color: ")

match color:
    case "red":
       print("Stop")
    case "yellow":
       print("Look")
    case "green":
       print("Go")
    case _:
      print("wrong color")
```

Simple value match for traffic lights

Practice Problems (Set 1)

1. For a given number `n`, print if it's a multiple of 5 or not.
2. For a given number `n`, print if it's Odd or Even.

Loops

A **loop** lets us **execute a block of code multiple times:** either a set number of times or until a condition is met.

Python has 2 main types of loops:

1. `while` loop - repeat while a condition is `True`
2. `for` loop - iterate over a sequence (like a list, string, or range)

The while Loops

Syntax of a `while` loop:

```
while condition:
    # code block
```

Whatever statements we write in the code block are all executed one after the other

Let's look at some examples:

```
# Example 1 - print 1 to 5
i = 1
while I <= 5:
    print(i)
    i += 1

# Example 2 - print 5 to 1
i = 5
while i > 0:
    print(i)
    i -= 1
```

If the loop expression is such that it always computes `True`, then such a loop never stops & is called an **infinite loop**. We should try to never unintentionally create such a loop.

```
while True: # DO NOT RUN this code
    print("Prime")
```

Example of an Infinite Loop

Loop control statements

Python gives us some tools to **control** how loops behave & a classical example of them are 2 statement - `break` & `continue`.

The `break` keyword

It stops the loop immediately.

```
# Break for multiple of 6
i = 1

while i <= 10:
    if(i % 6 == 0):
        break
    print(i)
    i += 1

# output: 1, 2, 3, 4, 5
```

The `continue` keyword

It skips the rest of the current iteration and moves to the next one.

```
# Skip multiples of 3
i = 0

while(i < 10):
    i += 1
    if(i % 3 == 0):
        continue;
    print(i)

# output: 1, 2, 4, 5, 7, 8, 10
```

The for Loops

In Python, a `for` loop is used to iterate (loop) over a sequence, such as a list, tuple, string, or range, and execute a block of code for each item in that sequence.

Syntax of a `for` loop:

```
for variable in sequence:
    # code to run for each item
```

Let's look at an example:

```
for i in range(5):
    print(i)

# output: 0, 1, 2, 3, 4
```

The `in` keyword that we used is a **membership operator** in Python, used to loop over a sequence or to check the presence of an item in a sequence, like a string.

```
word = "Prime"

# Example 1 - Looping over a string
for ch in word:
    print(ch)

# Example 2 - Check if char 'i' exists in word
if 'i' in word:
    print("letter exists")

# Example 3 - Count number of i's in the word
word = "artificial intelligence"
count = 0

for ch in word:
    if ch == 'i':
        count += 1

print(f"i occurs {count} times.")
```

Just like conditionals, we can also have **nested loops** i.e. loop inside a

```
for i in range(1, 3):
    for j in range(1, 3):
    print(f"({i}, {j})")

# output: (1, 2) (1, 2) (2, 1) (2, 2)
```

Nested Loop Example

The `range()` Function

The `range()` function in Python is used to generate a **sequence of numbers**, typically for use in loops. It doesn't actually create a list in memory (unless we convert it); instead, it creates an iterator that produces numbers one by one, which makes it very efficient.

The function has 3 parameters:

1. start (default=0) - the number to start from
2. stop - the number to stop before (not included)
3. step (default=1) - how much to increase by each time

```
# single argument - start
for i in range(5):
    print(i)

# output: 0, 1, 2, 3, 4

# 2 arguments - start, stop
for i in range(1, 6):
    print(i)

# output: 1, 2, 3, 4 , 5

# 3 arguments - start, stop, step
for i in range(1, 10, 2):
    print(i)

# output: 1, 3, 5, 7, 9
```

Practice Problems (Set 2)

1. Print multiplication table for any number `n` . [using `while`]
2. Print odd numbers from 1 to 10, using continue. [using `while`]
3. Count vowels in a word. [using `for`]
4. Sum of first `n` natural numbers. [using `for`]

Functions

Functions in Python are **blocks of reusable code** that perform a specific task. They make our code organized, readable, and easier to maintain.

Function Definition

We use the `def` keyword to define our function:

```python
# Function Definition
def hello():
    print("hello from Prime")
```

and to call the function i.e. to run it we write their name with parentheses:

```python
hello() # Function Call
```

Function call actually invokes the function & then all the block of code written in the functions definition is executed.

Functions with Parameters

We can also pass data to a function using **parameters** (inside the parentheses). Values actually passed when calling the function are called **arguments**.

```python
# Fnx to compute sum of 2 nums

def sum(a, b):          # a & b are parameters
    print (a + b)

print(sum(5, 10))       # 5 & 10 are arguments
```

wherever fnx → it means function

Functions can **return** a result using the `return` keyword.

```python
# Fnx to computer average of 3 nums

def avg(a, b, c):
    return (a + b + c) / 3
print(avg(1, 2, 3))
```

Default Parameters

We can assign **default values** to function parameters so that if the caller doesn't provide that argument, Python uses the default values. And these default parameters must come last. We cannot have a non-default argument after a default argument.

```python
# sum() fnx with default param 1
def sum(a, b = 1): # default val of b is 1
    return a + b

print(sum(5)) # output: 6
```

Built-in Functions

All the functions that we have studied till now are **user-defined functions**, but in Python we also have a lot of built-in functions too. The definition (logic) of **built-in functions** is already written in Python & we just have to use them.
Some built-in functions that we have already used till now:

1. `print()`
2. `input()`
3. `type()`
4. `range()`

Lambda Functions

Lambda functions are short, one-liner functions that are used to perform simple tasks. These are created using the `lambda` keyword. These functions are anonymous & do not have a name like regular functions defined with `def` .
Lambda function **Syntax**:

```
lambda arguments: expression
# arguments -> 1 or more parameters
# express ion -> single expression whose result is automatically returned
```

Let's see an example of a lambda function that computes square of a number:

```
# fnx to compute x^2
square = lambda x: x * x
print(square(5))
```

Practice Problems (Set 3)

1. Create a function to compute factorial of a number `n` .

2. Create a function to return largest of 3 numbers - `a` , `b` & `c` .

Solutions (Set 1 - Conditionals)

1. Multiple of 5 or not

```
num = int(input("enter number: "))

if num % 5 == 0:
    print("multiple of 5")
else:
    print("NOT a multiple of 5")
```

2. Odd or Even

```
num = int(input("enter number: "))

if num % 2 == 0:
    print("Even")
else:
    print("Odd")
```

Solutions (Set 2 - Loops)

1. Multiplication table of `n`

```
n = int(input("enter n: "))
i = 1

while i <= 10:
      print(i * n)
      i += 1
```

2. Odd nums from 1 to 10, using continue

```
i = 0

while(i < 10):
       i += 1
       if(i % 2 == 0):
              continue;
       print(i)

# output: 1, 3, 5, 7, 9
```

3. Count vowels

```
for ch in word:
    if (ch   == 'a' or ch == 'a' or ch == 'a' or ch == 'a' or ch == 'a'):
        count += 1

print(f"vowel count = {count}")
```

4. Sum of first n Natural numbers

```
n = int(input("enter n: "))
sum = 0
for i in range(1, n+1):
    sum += i

print("sum = ", sum)
```

Solutions (Set 3 - Functions)

1. Factorial of N

```
# Factorial of N
n = int(input("enter n: "))

fact = 1
for i in range (1, n+1):
fact *= i

print("factorial = ", fact)
```

2. Largest of 3 numbers

```
def get_largest (a, b, c):
    if (a > b and a > c):
        return a
    elif b > c:
        return b
    else:
        return c
print (get_largest (3, 10, 5))
```

Keep Learning & Keep Exploring!