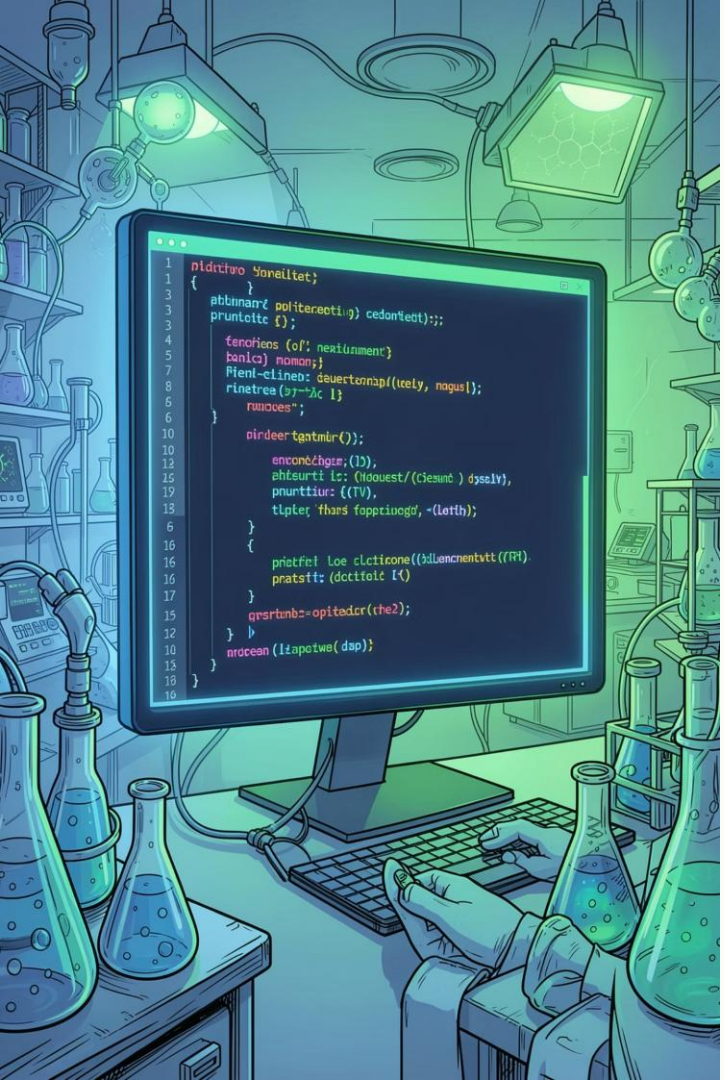


FACULTY DEVELOPMENT PROGRAMME

Program Logic: Control Structures & Modularity

Using Python to Automate Pharmaceutical Calculations

Deliver By:

Dr. Meenakshi Sharma
AICTE Post Doctoral Fellow, Tezpur University
Assistant Professor, CSE Department
Assam Kaziranga University (On Lien)

Learning Objectives

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

Understand Program Logic

What it is and why it matters

Use Decision-Making Statements

if, **else**, and **elif** in Python

Repeat Tasks Using Loops

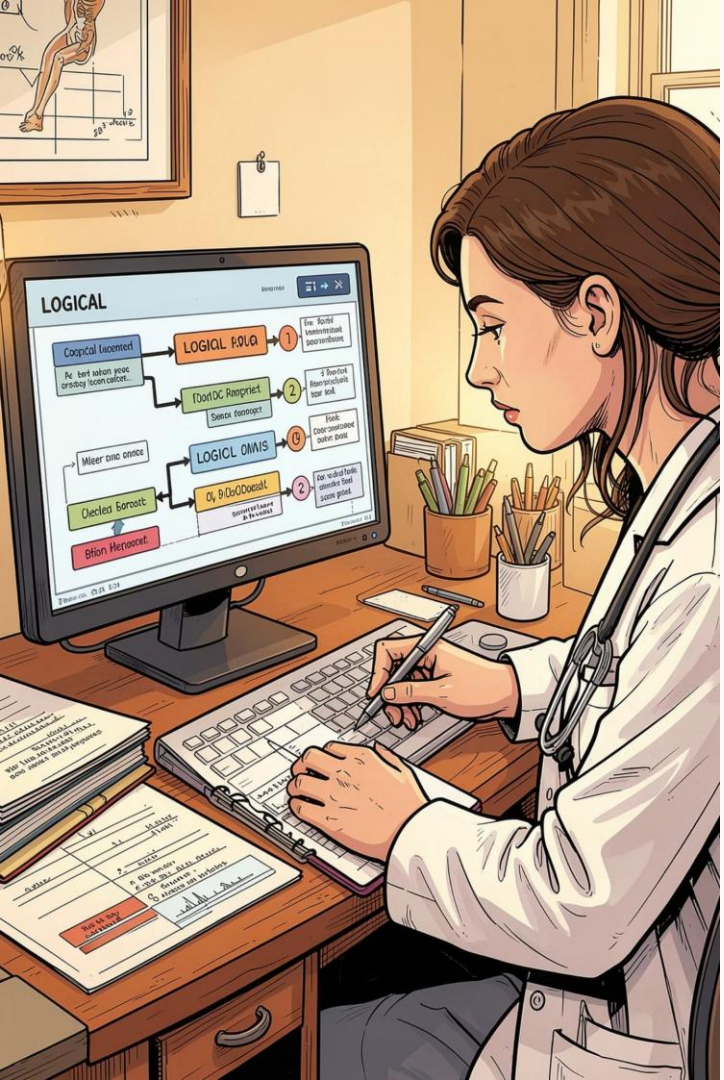
Automate repetitive pharmaceutical calculations

Create Reusable Functions

Write once, use many times

Apply to Pharmaceutical Problems

Real-world healthcare examples



What is Program Logic?

It is the **step-by-step sequence of instructions** that tells a computer what to do, when to do it, and how to do it to solve a problem.

🩺 A Doctor's Logic

Patient arrives → Collect symptoms → Perform tests → Analyze reports → Decide treatment

💻 A Computer's Logic

Receive input → Check conditions → Process data → Produce output → Take action

Key Message: A computer is intelligent only when we provide it with proper logic.

Why Program Logic Matters

❌ Without Program Logic

- Computers cannot make decisions
- Computers cannot automate repetitive work
- Computers cannot solve real-world problems

✓ With Program Logic

- Automatic patient screening
- Dosage calculations
- Clinical data analysis
- Drug research automation
- Medical decision support

ⓘ 💡 **Example:** Instead of calculating BMI for 500 patients manually, Python can do it in **seconds**.

Logic in Everyday Life & Programming

We already use logic every day. Programming follows the same pattern!

🌧️ Weather Decision

If it is raining → Carry an umbrella.

🩺 Blood Pressure Check

If blood pressure is high → Consult a physician.

🔬 Glucose Test

If glucose level > 200 → Patient is diabetic.

Programming follows the **same logic** — check a condition, then take action.

Three Building Blocks of Program Logic

Every computer program is built on three core concepts:

1. Decision Making

Choose one action among many based on a condition.

2. Repetition

Repeat a task multiple times without rewriting code.

3. Function/Modularity

Reuse code blocks whenever required.

These three concepts are the **foundation of almost every software application** in healthcare and beyond.

Decision Making in Python

Decision making allows Python to **choose different actions** depending on a condition.

"if" Statement (Check only true condition)

Python Syntax:

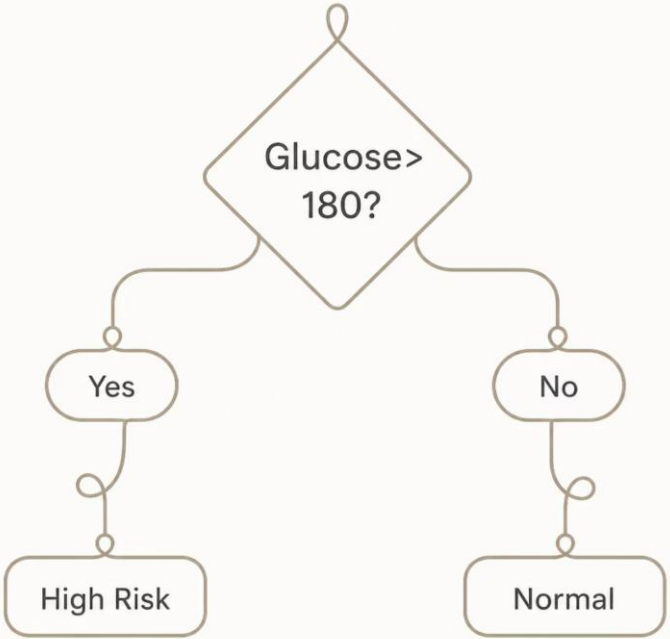
```
if condition check:
    statement
```

Example:

```
glucose = 220

if glucose > 180:
    print("Diabetic")
```

Output: Diabetic

Python checks the condition. If **True**, it executes the statement.



“if–else” & Multiple Decisions

“if–else” (check either true or false outcomes)

```
glucose = 140

if glucose > 180:
    print("High Risk")
else:
    print("Normal")
```

Output: Normal

“if–elif–else” (check more than two conditions)

```
glucose = 165

if glucose >= 200:
    print("Diabetic")
elif glucose >= 140:
    print("Prediabetic")
else:
    print("Normal")
```

Output: Prediabetic

Glucose Level	Category
< 140	Normal
140–199	Prediabetic
≥ 200	Diabetic

Application: Patient risk classification.

Nested “if” Statements

```
glucose = 175
age = 62

if age >= 60:
    if glucose >= 170:
        print("High Risk Patient")
```

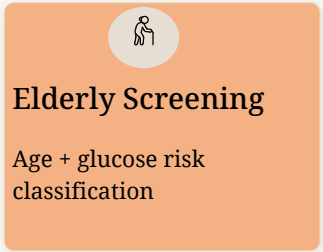
Output: **High Risk Patient**

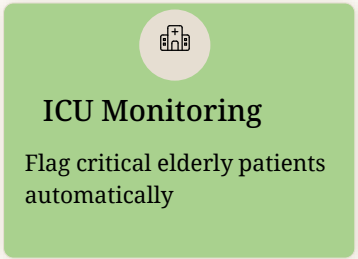
```
glucose = 135
age = 62

if age >= 60:
    if glucose >= 170:
        print("High Risk Patient")
    else:
        print("Age high but glucose level is Normal")
```

Output: **Age high but glucose level is Normal**

```
glucose = 125
age = 42

if age >= 60:
    if glucose >= 170:
        print("High Risk Patient")
    else:
        print("Age high but glucose level is Normal")
else:
    print("Age is below 60, so glucose test can not be done.")
```

Output: **Age is below 60, so glucose test can not be done.**





Python Programming: Getting Started with Google Colab

1. Open Google Chrome.
2. Type Google Colab or https://colab.research.google.com
3. After Colab opens, Click + **New Notebook**
    - A notebook is like a digital notebook where we write Python programs.
4. Understand the Colab Screen
    - At the top: we found Untitled0.ipynb file.
    - Rename it as: **Python_Basic.ipynb**
    - Here, Python_Basic is the file name
    - .ipynb = IPython Notebook file
        - . → Indicates a file extension.
        - ipy → interactive version of Python.
        - nb → Stands for Notebook.

5. Code Cell: Below the title, a box like [ ] is there which is known as Code Cell. Here Python Program cab be written. By clicking +**Code**, we can add more cells.

6. Run Button: On the left side of the code cell, ▶, run button is there, when we click this, program executes.

Why Do We Need Loops?

A **loop** repeatedly executes a block of code until a condition is met.

A pharmacist checks the glucose level of every patient in a list. Instead of checking one patient at a time manually, Python automatically repeats the same operation for every patient

It saves time and effort.

The “for” Loop

for loop is used when the **number of iterations is known**.

Process Patient Records

Iterate through a list of patient IDs

Calculate BMI

Compute BMI for every patient

Analyze Glucose Levels

Screen all values in a dataset

“for” Syntax:

```
for item in list:
```

for ->. The keyword that initiates the loop statement
item -> A temporary variable name you choose. It holds the value of the current element during each iteration.
in -> The keyword that connects your loop variable to the sequence.
list -> The list/data set where data are stored.

Example 1

```
# list of patient’s ID
patients = ["P1", "P2", "P3"]

# Display each patient’s ID
for i in patients:
    print(i)
```

Output: P1 P2 P3

The “for” Loop

Example 2

```
bmi_list = [22.04, 24.91, 22.03, 26.12, 24.09]

for bmi in bmi_list:
    print("BMI =", bmi)
```

Output:

```
BMI = 22.04
BMI = 24.91
BMI = 22.03
BMI = 26.12
BMI = 24.09
```

Example 3

```
bmi_list = [22.04, 24.91, 22.03, 26.12,24.09,
23.5, 21.6, 28.1, 22.9, 26.7]

for b in range(5):
    print(“BMI=”,bmi_list[b])
```

Output:

```
BMI = 22.04
BMI = 24.91
BMI = 22.03
BMI = 26.12
BMI = 24.09
```

The `while` loop

A `while` loop repeats until a given condition becomes `False`.

“while” Syntax:

```
while condition:
        statements execute
    update by increment/decrement
```

```
# Patient body temperature (°C)
temperature = 35

# Continue checking while temp. is above 38°C
while temperature < 38:
        print(“Normal")
        temperature = temperature + 1

print(“Patient has fever")
```

Key points: While condition must be false at some point, otherwise it enters in an infinite loop.

Output:

Normal
Normal
Normal
Patient has fever

- First check: **35 < 38** → True → prints **Normal**, temperature becomes **36**
- Second check: **36 < 38** → True → prints **Normal**, temperature becomes **37**
- Third check: **37 > 38** → True → prints **Normal**, temperature becomes **38**
- Forth check: **38 > 38** → False → loop stops.

Application: Monitor heart rate continuously until a specified condition met.

The break & continue : jump statements used in loops

A break statement terminate an ongoing loop if specified emergency condition meets.

```
temperature = 38
while temperature <= 42:
    print("Temperature =",
temperature, "°C -> Fever")

    if temperature == 40:
        print("Emergency! No need
        to check further.")
        break

    temperature = temperature + 1
```

Output:

Temperature = 38 °C -> Fever
Temperature = 39 °C -> Fever
Temperature = 40 °C -> Fever
Emergency! No need to check further.

A Continue statements skips the current iteration and force the loop to execute next iteration when certain condition meets

```
temperature = 33
while temperature <= 37:

    if temperature == 35:
        print("Sensor Error! Skip this...")
        temperature = temperature + 1
        continue

print("Temperature =", temperature, "°C -> No Fever")
temperature = temperature + 1
```

Output:

Temperature = 33 °C -> No Fever
Temperature = 34 °C -> No Fever
Sensor Error! Skip this...
Temperature = 36 °C -> No Fever

for vs. while vs. break vs. continue: Which to Use?

Feature	for Loop	while Loop	break	continue
Purpose	Repeat for a fixed number of items	Repeat while a condition is true	Exit the loop immediately	Skip the current iteration and continue
Iterations	Usually known	Usually unknown	Stops loop early	Continues to next iteration
Stops When	All items processed	Condition becomes False	break statement executes	Current iteration ends
Best For	Lists, datasets	Continuous monitoring	Emergency stop	Ignoring unwanted data
Pharmacy Example	Process patient records	Monitor patient's temperature until normal	Stop when critical glucose is detected	Skip incomplete patient records

for

Repeat every item

while

Repeat until condition changes

break

Exit loop immediately

continue

Skip current iteration

Tip: Use the right tool for the right situation to write efficient, clear, and safe code.

Functions and Modularity

A **function** is a reusable block of code created by the programmer to perform a specific task. Instead of writing the same code repeatedly, you define it once and call it whenever needed.

Example: like a lab instrument that performs one task reliably every time, when it is used.

Saves Time

Call the same code anywhere

Easy Maintenance

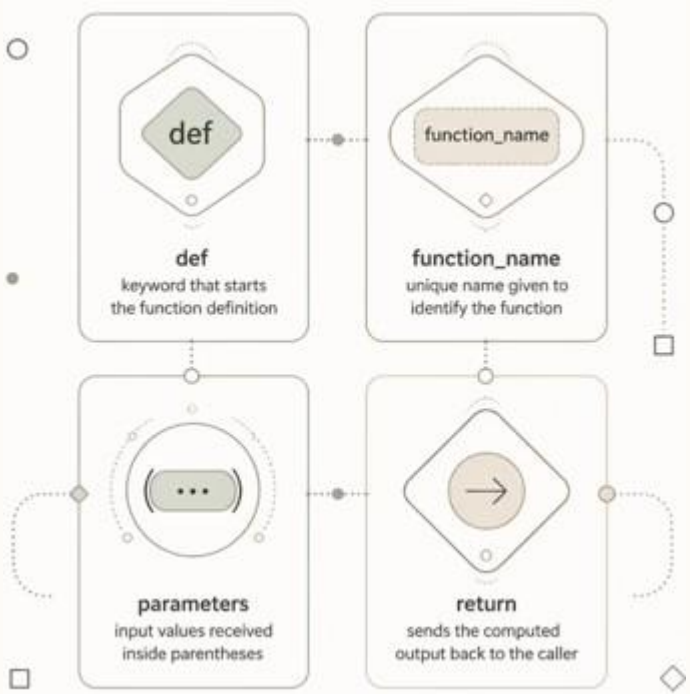
Update once, applies everywhere

Fewer Mistakes

Less copying means fewer errors

Function Structure in Python

```
def function_name(parameters):
    statements
    return result to the caller.
```



What Each Component Does

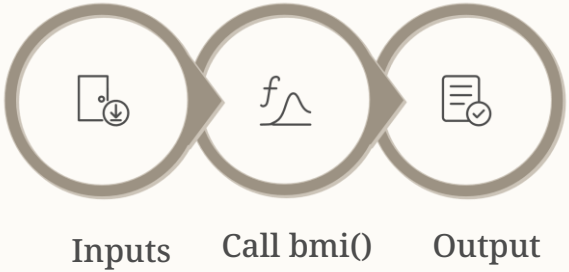
- **def** — Keyword that signals the start of a function definition
- **function_name** — A unique identifier/name you assign to the function
- **parameters** — Input values the function receives when called
- **return** — Sends the computed result back to the caller

Pharmaceutical Example: BMI Calculator

```
def bmi(weight, height):
    bmi_value = weight / (height ** 2)

    return bmi_value

result = bmi(65, 1.65)

print("BMI=",result)
```

Output: BMI=23.88 — a healthy BMI range

How the BMI Function Executes



- Inputs flow into the bmi function
- Calculation is performed using formula in it
- Result is returned to the caller
- Result is displayed

Parameters: The Inputs to a Function

Parameters are the input values a function receives when it is called.

```
bmi (weight, height)
```

Inside the Function

```
def bmi(weight, height):
    bmi_value = weight / (height ** 2)
```

Calculate bmi_value for each input given by the user

```
P1 = bmi(65, 1.65)
```

```
P2 = bmi(68, 1.45)
```

```
P3 = bmi(70, 1.49)
```

Return Values: Sending Results Back

Return by the Function

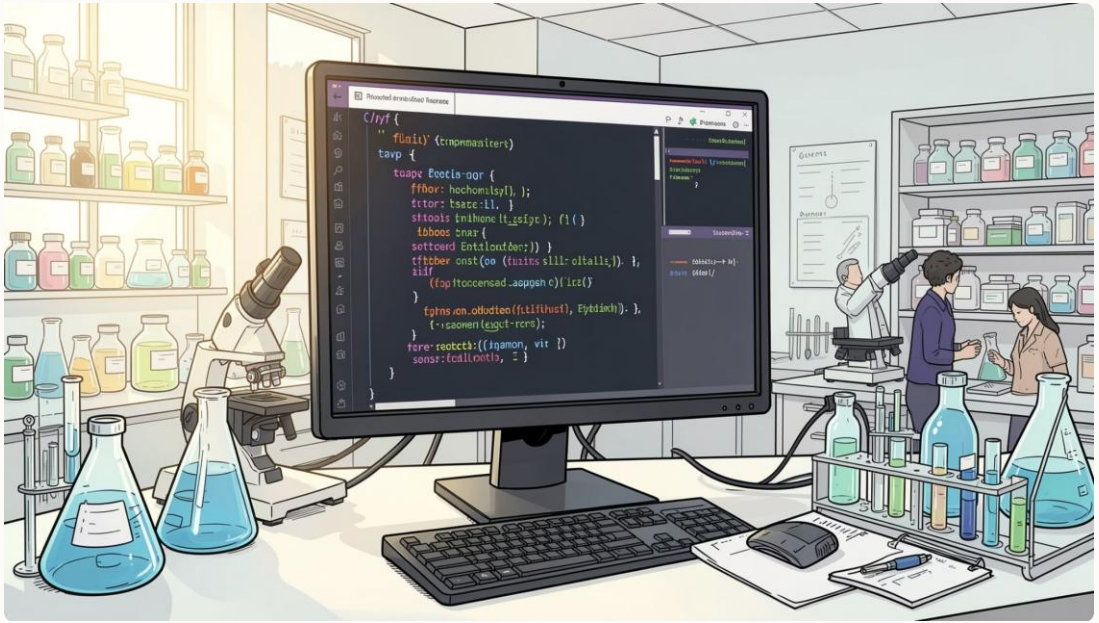
```
return bmi_value
```

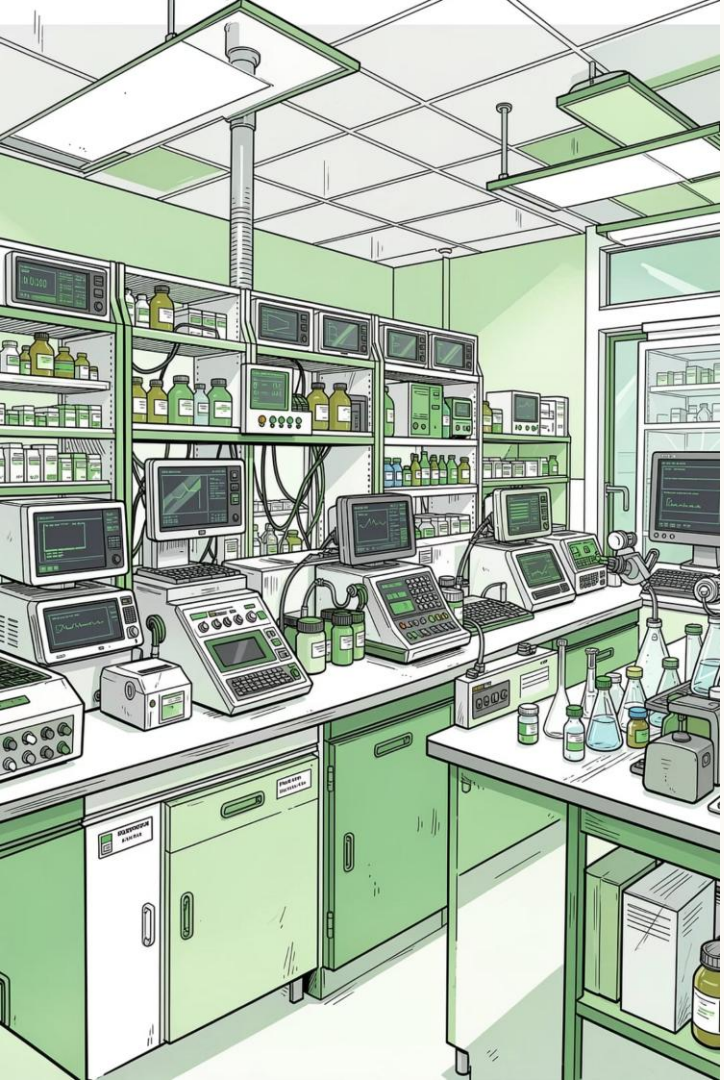
Sends the computed result back to the caller (who called the function).

Storing the Result

```
result = bmi(65, 1.65)
print(result)
```

The returned value is stored in a variable named result for further use.





Why Use `return`?

Reuse the Value

Store and use the calculated result multiple times without recomputing

Further Calculations

Feed the returned value into other functions or formulas downstream

In the considered example:

- The returned BMI value can be stored and later used to determine whether the patient is underweight, normal, overweight, or obese.
- The BMI returned by the function is used in another calculation to decide the appropriate medication dose.

Complete Example 1: Display BMI and weight Category

```python
# Function to calculate BMI
def bmi(weight, height):
    bmi_value = weight / (height ** 2)
    return bmi_value

# Input values
weight = 65      # in kilograms
height = 1.65    # in meters

# Function call/ Caller
result = bmi(weight, height)

# Display BMI up to 2 decimal point
print("BMI =", round(result, 2))

# BMI Classification
if result < 18.5:
    print("Category: Underweight")
elif result < 25:
    print("Category: Normal Weight")
elif result < 30:
    print("Category: Overweight")
else:
    print("Category: Obese")
```

Output:

```
BMI = 23.88
Category: Normal Weight
```

Complete Example 2: Display BMI and Dosage

```python
# Function to calculate BMI
def bmi(weight, height):
    bmi_value = weight / (height ** 2)
    return bmi_value

# Function to calculate dosage
def dosage(weight):
    dosage_value = weight * 2
    return dosage_value

# Input values
weight = 65      # in kilograms
height = 1.65    # in meters

# Function calls / Callers
bmi_value = bmi(weight, height)
dosage_value = dosage(weight)

# Display results
print("BMI =", round(bmi_value, 2))
print("Dose =", dosage_value, "mg")
```

bmi function

Dosage function

Output:

```
BMI = 23.88
Dose = 130 mg
```

Benefits of Modular Script Architecture for Pharmaceutical Calculations

Easy to Understand

Each function has a clear, single purpose — readable by pharmacists and programmers alike.

Easy to Maintain

Update one function without affecting the rest of the program.

Reusable

Call the same function across multiple programs and patient cases.

Fewer Errors

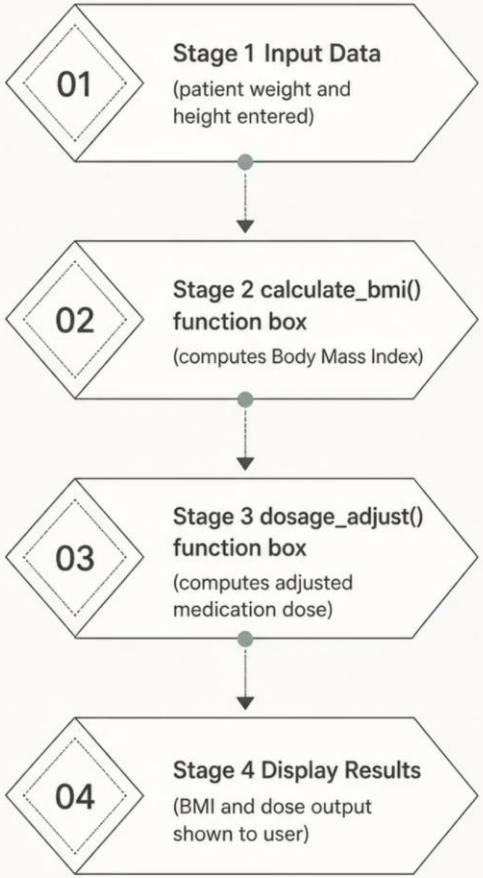
Isolated functions reduce the risk of calculation mistakes in critical dosage scripts.

Sample Script Architecture

A well-structured pharmaceutical script flows through distinct stages — from data input through dedicated calculation functions to final output.

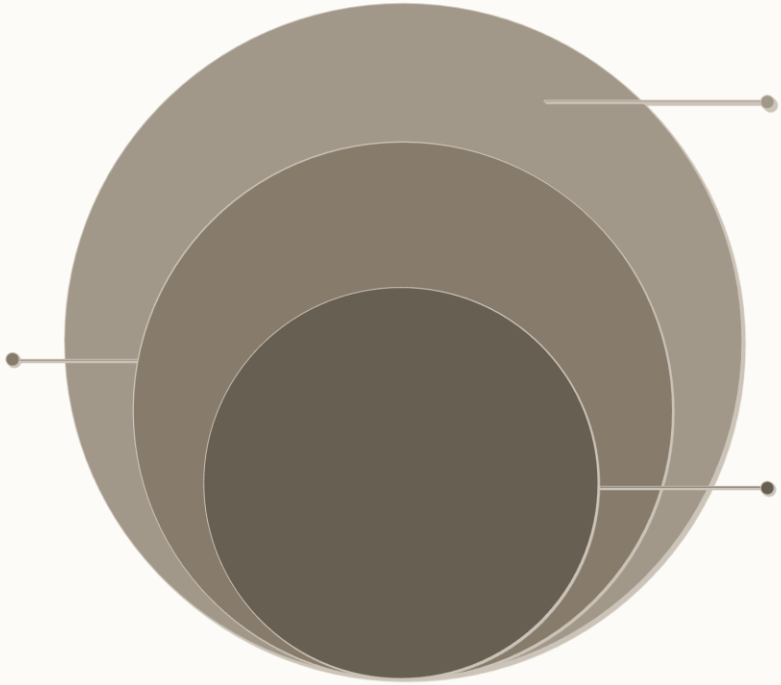
- **Input Stage** — Patient weight and height captured
- **calculate_bmi()** — Returns BMI value
- **dosage_adjust()** — Returns adjusted dose in mg
- **Display Stage** — Display both BMI and Dosage for review

Each function performs **one specific task** — keeping calculations transparent and auditable.



This modular approach ensures pharmaceutical scripts are accurate, maintainable, and aligned with professional programming standards.

Learning Outcome



Output to users

print() displays results to user

Calculation Functions/ Modules

bmi() and dosage() compute values

Python Main Program

Main script coordinating functions

This modular approach ensures pharmaceutical scripts are accurate, maintainable, and aligned with professional programming standards.

Pharmaceutical Applications of Programing Logic

Dosage Calculations

Automate weight-based and renal dosing.

Patient Screening

Classify risk using if-elif-else logic

Batch Processing

Process hundreds of records with loops

Clinical Data Analysis

Analyze trial data and generate reports

Key Takeaway: Program logic transforms manual, error-prone tasks into fast, reliable automated workflows.

Thank You

For Your Attention!

Any Questions

Python : Decision Making and Looping

Tushar B. Kute,
http://tusharkute.com

Decision Making Statement

if-elif-else

1

```
if condition:
    statements
```

2

```
if condition:
    Statement-1
else:
    Statement-2
```

3

```
if condition:
    statement-1
elif condition:
    Statement-2
else:
    Statement-3
```

if-else

```
num = 23
if num > 0:
    print('Positive Number')
else:
    print('Negative Number')
```

Previous line of indentation

Indentation

if-elif-else

```
num = -85
if num == 0:      # Compulsory
    print('It is ZERO')
elif num > 0:     # Optional, Repeatable
    print('Positive Number')
else:             # Optional
    print('Negative Number')
```

Condition in if-else

- The condition in if-statement is parenthesis free.
- The condition can have ‘and’ and ‘or’ operator.
- It should end with a colon.
- It must return either True or False.
- The zero value is considered as False otherwise True.
  - Example:

    if True:

Exercise

- Write a program to read a number from user and find whether number is odd or even.

Nesting of if-else

- Example:
  - Read three numbers from keyboard and find largest of them. You are not allowed to use ‘and’ and ‘or’ operators.

Example:

```
num1 = int(input("Enter first:"))
num2 = int(input("Enter second:"))
num3 = int(input("Enter third:"))
if num1 > num2:
    if num1 > num3:
        print ("Largest is", num1)
    else:
        print ("Largest is", num3)
else
    if num2 > num3:
        print ("Largest is", num2)
    else:
        print ("Largest is", num3)
```

Example:

- Read an year and find whether that year is leap or not?
- Conditions for being leap year:
  - It should be divisible by four.
  - If its a century year, it should be divisible by 400 also.

Solution:

```
year = int(input("Enter the year: "))
if year % 4 == 0:
    if year % 100 == 0:
        if year % 400 == 0:
            print ('Leap')
        else:
            print ('NOT Leap')
    else:
        print ('Leap')
else:
    print ('NOT Leap')
```

Using ‘and’ and ‘or’

```
year = int(input("Enter the year: "))
if year%4==0 and year%100!=0 or year%400 ==0:
    print('LEAP')
else:
    print('NOT Leap')
```

Repeatable elif

- Read marks in percentages from student and find the grade of it as per below condition:
  - Above 75% : ‘Distinction’
  - 60% to below 75%: ‘First Class’
  - 50% to below 60%: ‘Second Class’
  - 40% to below 50%: ‘Pass Class’
  - Below 40%: ‘Failed’

Solution

```
marks = float(input('Enter the marks: '))

if marks >= 75:
    print('Distinction')
elif marks >= 60:
    print('First Class')
elif marks >= 50:
    print('Second Class')
elif marks >= 40:
    print('Pass Class')
else:
    print('FAILED')
```

Another way

```
marks = float(input('Enter the marks: '))

if marks < 40:
    print('FAILED')
elif marks < 50:
    print('Pass Class')
elif marks < 60:
    print('Second Class')
elif marks < 75:
    print('First Class')
else:
    print('Distinction')
```

Creating menu based programs

- Python does not have switch-case statement. So menu based programs are created simply using if-elif-else.
- Example:
  - Read a number from user and do the following by showing menu:
    - Find its square
    - Find cube
    - Check odd / even
    - Check positive / negative

Solution:

```
num = int(input("Enter the number: "))
print ("* Menu *\n1.Square\n2.Cube\n3.Odd\n4.+ve")
ch = int(input("Enter the choice: "))
if ch == 1:
    print ("Square:", num * num)
elif ch == 2:
    print ("Cube:", num * num * num)
elif ch == 3:
    if num % 2 == 0:
        print ("Even")
    else:
        print ("Odd")
elif ch == 4:
    if num > 0:
        print ("+ve")
    else:
        print ("-ve")
else:
    print ("Wrong Choice")
```

Exercises

- Write a Python program to calculate profit or loss. Input is selling cost and actual cost.
- Write a Python program to check whether a character is uppercase or lowercase alphabet.
- Write a Python program to input electricity unit charges and calculate total electricity bill according to the given condition:
  - For first 50 units Rs. 0.50/unit
  - For next 100 units Rs. 0.75/unit
  - For next 100 units Rs. 1.25/unit
  - For unit above 250 Rs. 1.50/unit
  - An additional surcharge of 17% is added to the bill

Exercises

- **Input -> basic salary**
- **Output -> gross salary**

  DA = 75% of basic

  HRA = 20% of basic
- **Conditions:**

  < 10000 : gross = da + basic

  >= 10000 and < 20000 : gross = da + basic + 50% of hra

  >= 20000 : gross = basic + da + hra
- **Sample: Input and Output**

  5000 : 3750 + 5000 = 8750

  12000 : 9000 + 12000 + 1200 = 22200

  24000 : 18000 + 24000 + 4800 = 46800

Loops


Python
Loops
while loops
for loops
Nested Loops
Loop Control Statements

The while loop

```
while condition:          #compulsory
   statement1
   statement2
else:                     #optional
   statements
```

Example-1

- Print your name for 10 number of times.

```
count = 0                # initialize
while count < 10:        # condition
    print('Tushar')
    count += 1           # change counter
```

Exercise

- Find addition of first ten natural numbers. That is, find addition of numbers from 1 to 10.

Solution

```
count, add = 1, 0
while count <= 10:
    add = add + count
    count += 1
else:
    print('Addition is', add)
```

```
Addition is 55
```

Exercise

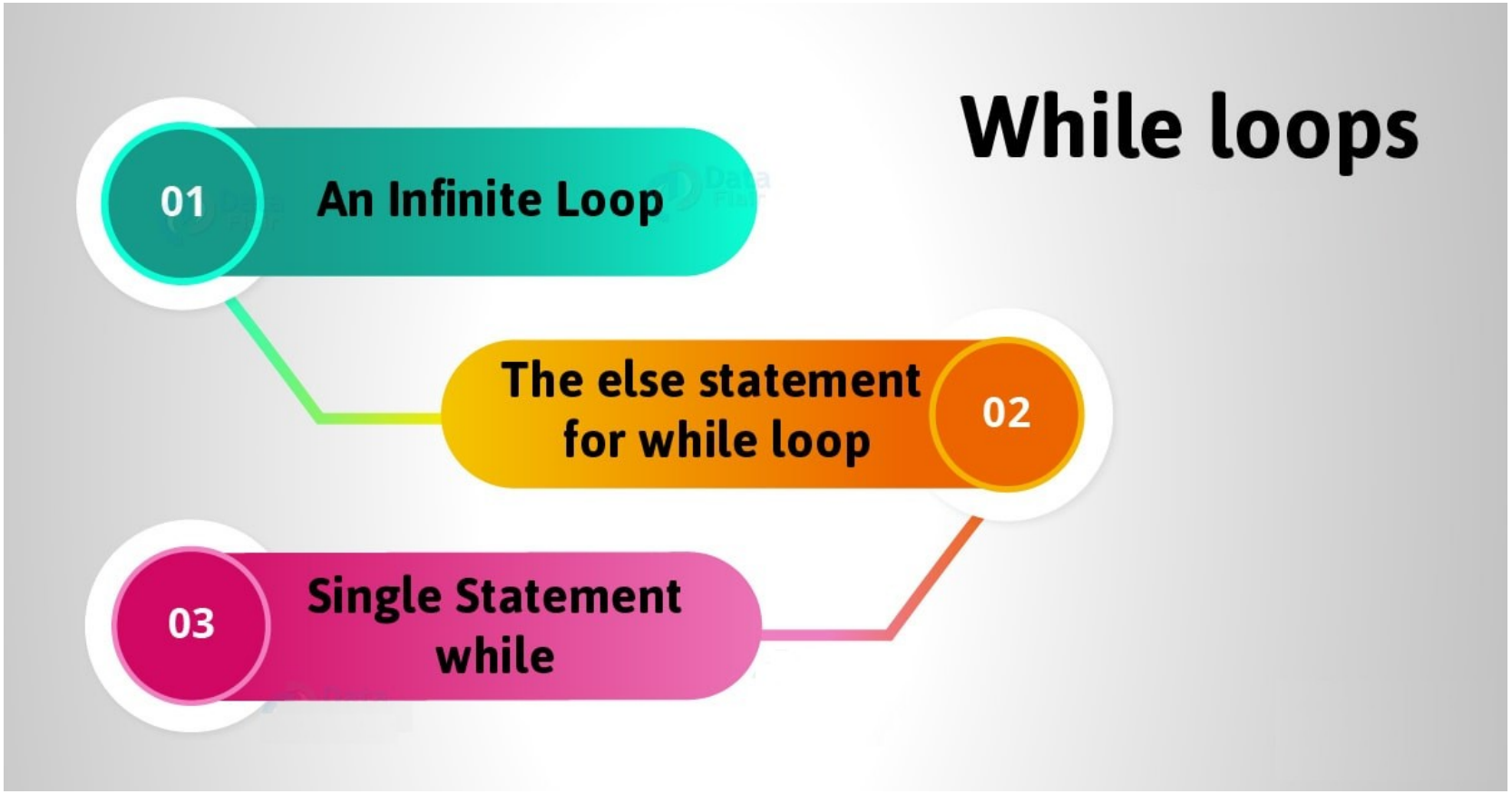
- Find addition of all the odd numbers from 1 to 20. That is, find addition of numbers 1, 3, 5 … 19.

Solution

```
count, add = 1, 0
while count <= 20:
    add = add + count
    count += 2
else:
    print('Addition is', add)
```

```
Addition is 100
```

More about ‘while’



Infinite loop

- Be careful while using a while loop. Because if you forget to increment the counter variable in python, or write flawed logic, the condition may never become false.
- In such a case, the loop will run infinitely, and the conditions after the loop will starve.
- To stop execution, press Ctrl+C. However, an infinite loop may actually be useful.
- While is the only loop where we can make while loop intentionally infinite.

Examples:

```
num = 0
while num < 10:
    print(num)

# Counter change missing
print('Ended')
```

1

2

```
while True:
    print('Tushar')
```

The else statement

- A while loop may have an else statement after it. When the condition becomes false, the block under the else statement is executed.
- However, it doesn't execute if you break out of the loop or if an exception is raised.

```
num = 0
while num < 10:
    print(num)
    num += 1
else:
    print('Ended')
```

Single statement while

- We can write single statement in while loop. This statements will be separated by semicolon.

```
num = 0
while num < 10: print(num); num += 1
```

The ‘for’ loop

- Python for loop can iterate over a sequence or collection of items.
- The structure of a for loop in Python is different than that in C++ or Java.
- That is, for(int i=0;i<n;i++) won’t work here. In Python, we use the ‘in’ keyword.
- Syntax:

```
for n in sequence:
    statements
```

n : variable
sequence: collection of elements

while vs for loop

while loop	for loop
Can have condition for iteration	Don’t have any loop condition
The counter can be incremented or decremented by any number	The loop can iterate through sequence by one increment only
The loop can be infinite	This loop can’t be infinite

Examples:

```
>>> for n in 1,6,7,8,4:
...     print(n, end=' ')
...
1 6 7 8 4
>>> for n in 1,6,7,8,4:
...     print(n*n, end=' ')
...
1 36 49 64 16
>>> for n in 'h',23,3.12,12:
...     print(n, end=' ')
...
h 23 3.12 12
```

Same sequence

Operation on data

Mixed Elements

More on ‘for’ loop

The range( ) function

- This function yields a sequence of numbers. When called with one argument, say n, it creates a sequence of numbers from 0 to n-1.
- Example:

```
>>> arr = range(10)
>>> list(arr)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr = range(1,6)
>>> list(arr)
[1, 2, 3, 4, 5]
>>> arr = range(1,12,2)
```

Using range( ) function

```
>>> for x in range(10):
        print(x, end=' ')

0 1 2 3 4 5 6 7 8 9
>>> for x in range(1,10,2):
        print(x, end=' ')

1 3 5 7 9
```

The range( ) decrement operation

```
>>> arr = range(12,2,-2)
>>> list(arr)
[12, 10, 8, 6, 4]
```

Iterating through sequence

```
X = [4,7,9,1,5]          # list
>>> for num in x:
      print(num, end=' ')

4 7 9 1 5
>>> s = 'MITU Skillologies'    # string
>>> for data in s:
      print(data, end=' ')

M I T U   S k i l l o l o g i e s
```

Collection of strings

```
>>> col = ['Mar','Hin','San','Nep']
>>> for name in col:
        print(name)
Mar
Hin
San
Nep
```

The else statement

- Like a while loop, a for-loop may also have an else statement after it. When the loop is exhausted, the block under the else statement executes.

```
>>> for x in range(10):
        print(x, end=' ')
    else:
        print('Loop Ended')

0 1 2 3 4 5 6 7 8 9 Loop Ended
```

Example:

- Print your name 10 times using for loop.

```
for n in range(10):
    print('Tushar')
```

Example:

- Find addition of first 10 natural numbers using for loop.

```
add = 0
for n in range(1,11):
    add += n
else:
    print('Sum:',add)
```

Example:

- Find addition of all odd numbers from 1 to 20 using for loop.

```
add = 0
for n in range(1,21,2):
    add += n
else:
    print('Sum:',add)
```

Exercises

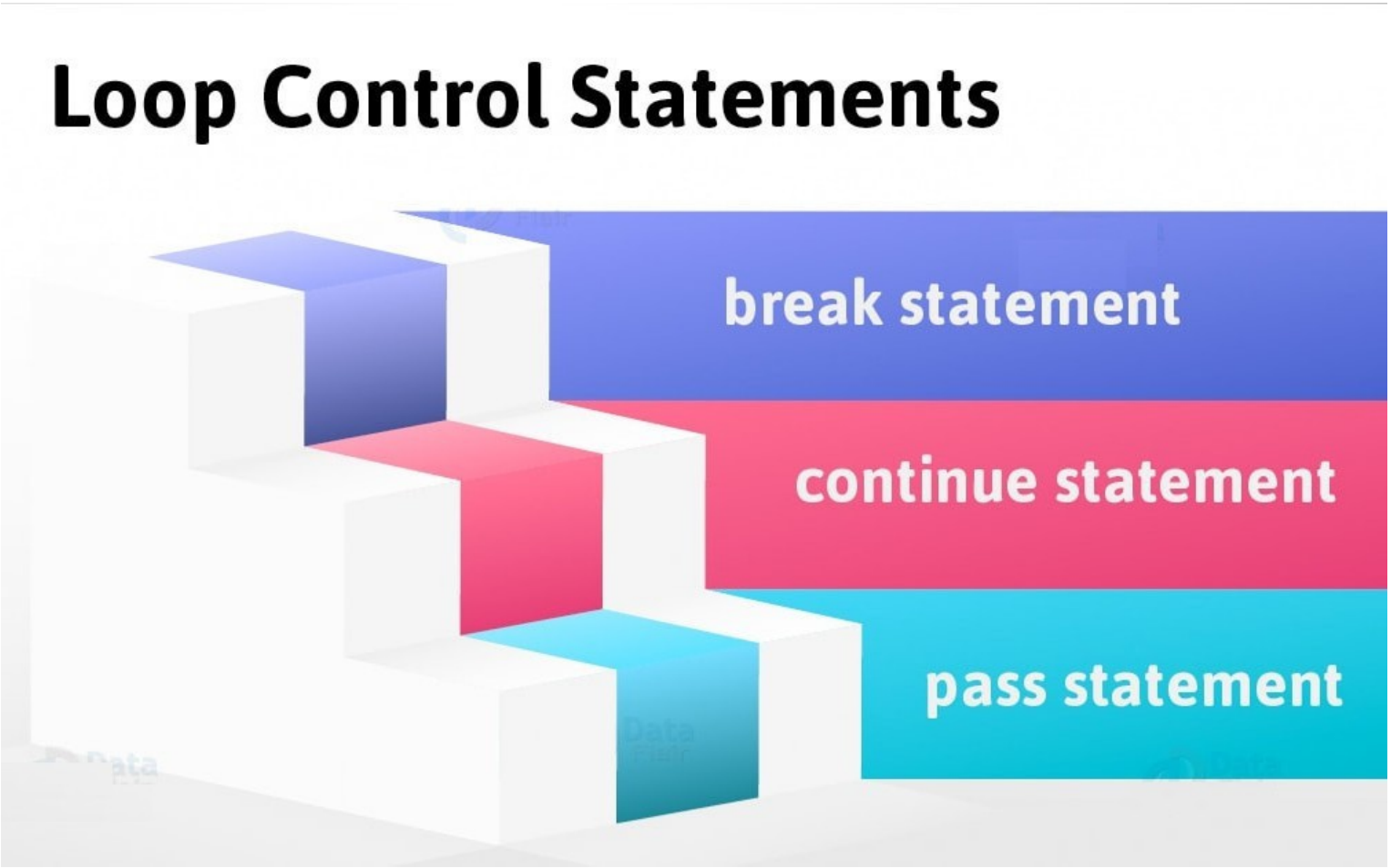
- Read a number from user and find the factorial of the number.
- Take a number user input and find sum of digits of this number.
- Write a program to print all the odd numbers from 10 to 30.
- Read a number from keyboard and print it in reverse.
- Write a program to print Fibonacci series up to n terms.

Nesting of loops

- You can also nest a loop inside another.
- You can put a for loop inside a while, or a while inside a for, or a for inside a for, or a while inside a while.
- Or you can put a loop inside a loop inside a loop. You can go as far as you want.

```
for i in range(1,6):
    for j in range(i):
        print("*",end=' ')
    print()
```

Nesting of loops



The ‘break’ statement

- When you put a break statement in the body of a loop, the loop stops executing, and control shifts to the first statement outside it. You can put it in a for or while loop.

```
for i in range(10):
    if i == 5:
        break
```

The ‘continue’ statement

- When the program control reaches the continue statement, it skips the statements after ‘continue’.
- It then shifts to the next item in the sequence and executes the block of code for it. You can use it with both for and while loops.

```
for i in range(10):
    if i % 2 == 0:
        continue
    print(i, end=' ')
```

```
1 3 5 7 9
```

The ‘pass’ statement

- We use the pass statement to implement stubs.
- When we need a particular loop, class, or function in our program, but don’t know what goes in it, we place the pass statement in it.
- It is a null statement.
- The interpreter does not ignore it, but it performs a no-operation (NOP).
- It is used to fill indented space

```
while True:
    pass
```

Exercises

- Write a program to check whether entered number if prime or not.
- Write a program to print all the prime numbers from 5 to 50.
- Print following pattern:

  1

  2 2

  3 3 3

  4 4 4 4

  5 5 5 5 5
- Find power of x raised to y from user input.

Thank you

This presentation is created using LibreOffice Impress 5.1.6.2, can be used freely as per GNU General Public License

@mitu_skillologies

/mITuSkillologies

@mitu_group

/company/mitu-skillologies

c/MITUSkillologies

Web Resources
http://mitu.co.in
http://tusharkute.com

contact@mitu.co.in

tushar@tusharkute.com

Loops and Conditionals

HORT 59000

Lecture 11

Instructor: Kranthi Varala

Relational Operators

- These operators compare the value of two ‘expressions’ and returns a Boolean value.

```
>>> A = 10
>>> B = 10
>>> A == B
True
```

```
>>> A = 10.0
>>> B = 10
>>> A == B
True
```

```
>>> A = '10'
>>> B = 10
>>> A == B
False
```

- Beware of comparing across data types, especially when reading values in from command line or files.

Relational Operators

==	equal	True if expressions are equal
!=	not equal	True if expressions are not equal
>	Greater than	True if left is greater than the right
<	Less than	True if left is less than the right
>=	greater than OR equal	
<=	less than OR equal	
is	identity	True if the left is the same object as right
in	contains	True if the object on left is contained in object on right (Useful for finding values in list)

Assignment Operators

- A += B increase A by value of B
- A -= B decrease A by value of B
- A *= B multiply A by B and assign value to A
- A /= B divide A by B and assign value to A
- A **=B raise value of A to the power of B
- A %= B modulus of A by B, assigned to A
- A //= B floor of A divided by B, assigned to A

- String context:
  - S1 += S2 add string on right to the one on left
  - S1 *= A Make A copies of S1 and concatenate them to S1

Boolean Operators

Combines two or more statements that return a Boolean value.

A and B	True if both A and B are true
A or B	True if either A or B is true
not A	reverse the Boolean given by A
xor(A,B)	True if only one of A or B is True

A	B	A and B	A or B	Not A	xor(A,B)
TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE	FALSE

General Python Syntax rules

- End of line is end of statement
- Statements at the same indentation level are in the same block (e.g., within a loop or condition)
- End of indentation
- Exceptions:
  - Semi colon ; separates statements on the same line

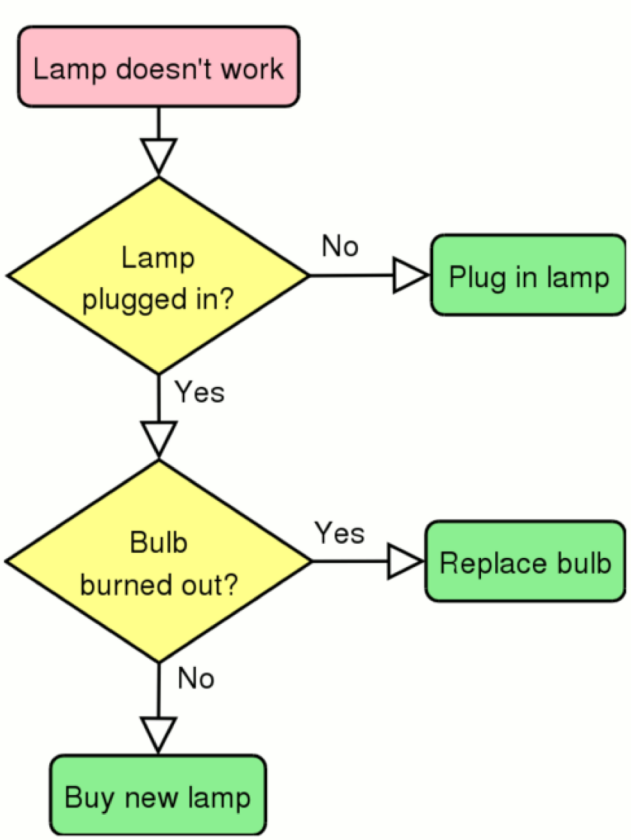
```
>>> A = 10 ; B =20 ; A == B
False
```

  - Single line blocks are allowed without indentation

```
>>> for x in range(5): print x
...
0
1
2
3
4
```

Branching logic

- Used to implement alternate paths for the logic flow.



https://upload.wikimedia.org/wikipedia/commons/4/44/LampFlowchart.png

If/elif/else statements

```
if test1:
    statement 1
elif test2:
    statement 2
else:
    statement 3
```

- Both the elif and else blocks are optional.

If/elif/else statements

```
>>> A = 0; B = 10; S1 = 'Hello'
>>> if (A < B ):
...     print 'true'
...
true
>>> if (A > B ):
...     print 'true'
... else:
...     print 'false'
...
false
```

```
>>> if A == 0 :
...     print A
... elif B ==10 :
...     print B
... else:
...     print "Neither match"
...
0
```

Lamp flowchart with if/else

```
#!/usr/bin/python

lamp = raw_input("Is the lamp on (yes/no): ")
plugged = raw_input("Is the lamp plugged in (yes/no) : ")
burnt = raw_input("Is the bulb burnt out (yes/no) : ")
if lamp != 'yes':
        if plugged == 'yes':
                if burnt == 'yes':
                        print 'replace bulb'
                else:
                        print 'replace lamp'
        else:
                print 'plug in lamp'
else: print 'Enjoy the light'
```

Accepts input from user, as a string

Truth and Boolean tests in Python

- All objects in python have an inherent true or false value.
- Any nonempty object is true.
  - For Integers : Any non-zero number is true
- Zero, empty objects and special object ‘None’ are false.
- Comparisons return the values True or False

Loops/Iterations

- A loop is syntax structure that repeats all the statements within the loop until the exit condition is met.
- Statements in a loop are defined by indenting them relative to the loop start.
- Loop ends when indentation ends.
- Python has two forms of loops: for loop and while loop.
- E.g. >>> for x in range(10)
- E.g. >>>while (A==10)

while loops

- while condition:
  statement 1
  statement 2
  .
  .

- Most generic form of loop, that checks whether the condition is true at the start of each iteration.
- Expects the condition to become false at some point during the iterations of the loop.
- If condition is never changed, this creates an ‘infinite’ loop. i.e., the program will get stuck in this loop for ever.

Example while loops

```
>>> while (A < B):
...     print A
...     A+=1
...
0
1
2
3
4
5
6
7
8
9
```

```
>>> while (B > A):
...     print B
...     B-=1
...
10
9
8
7
6
5
4
3
2
1
```

```
>>> while S1:
...     print S1
...     S1 = S1[1:]
...
Hello
ello
llo
lo
o
```

Altering while loops

- Normal loop control will execute all statements in block on every iteration. Loop ends only when exit condition is met.
- break statement forces the current loop to exit.
- continue statement skips the rest of the block and goes to the next iteration of the loop.
- pass statement is a placeholder for empty blocks.

Altering while loops

```
>>> A=0
>>> while True:
...     print A
...     if (A >= 5): break
...     A+=1
...
0
1
2
3
4
5
```

```
>>> A=0
>>> while A <= 5 :
...     A+=1
...     if (A == 3):continue
...     print A
...
1
2
4
5
6
```

for loops

- for item in sequence:
  statement 1
  statement 2
  .
  .

- Generic iterator for items in a ordered sequence such as lists, tuples etc.
- On each iteration retrieves one item from the list and assigns it to the variable specified.
- Automatically moves to the next item in the order.
- Value of variable may be altered within the for loop, but change is not made in the list.

for loops

```
>>> L=['Egg','Bacon','Ham']
>>> for item in L:
...     item+='s'
...     print item
...
Eggs
Bacons
Hams
>>> L
['Egg', 'Bacon', 'Ham']
```

```
>>> for x in range(0,100,10):
...     print x
...
0
10
20
30
40
50
60
70
80
90
```

Looping over Strings and Lists

- List is a general sequence object while String is a character sequence object.
- Both can be iterated over by a for loop:

```
>>> L
['Egg', 'Bacon', 'Ham']
>>> for x in L:
...     print x
...
Egg
Bacon
Ham
```

```
>>> S1="Hello"
>>> for x in S1:
...     print x
...
H
e
l
l
o
```

Looping over lists with and without index

- Looping with an index allows accessing the item within the list and changing it.

```
>>> for x in L:
...     x+='s'
...     print x
...
Eggs
Bacons
Hams
>>> L
['Egg', 'Bacon', 'Ham']
```

```
>>> for x in range(len(L)):
...     L[x]+='s'
...     print L[x]
...
Eggs
Bacons
Hams
>>> L
['Eggs', 'Bacons', 'Hams']
```

Looping over Tuples and Dictionaries

```
>>> T = (1,'a',45,23.45,'String')
>>> for x in T: print x
...
1
a
45
23.45
String
```

```
>>> myDict = {'a':'1','b':'2','c':'3'}
>>> for key in myDict:
...     print(key,' ==> ', myDict[key])
...
('a', ' ==> ', '1')
('c', ' ==> ', '3')
('b', ' ==> ', '2')
```

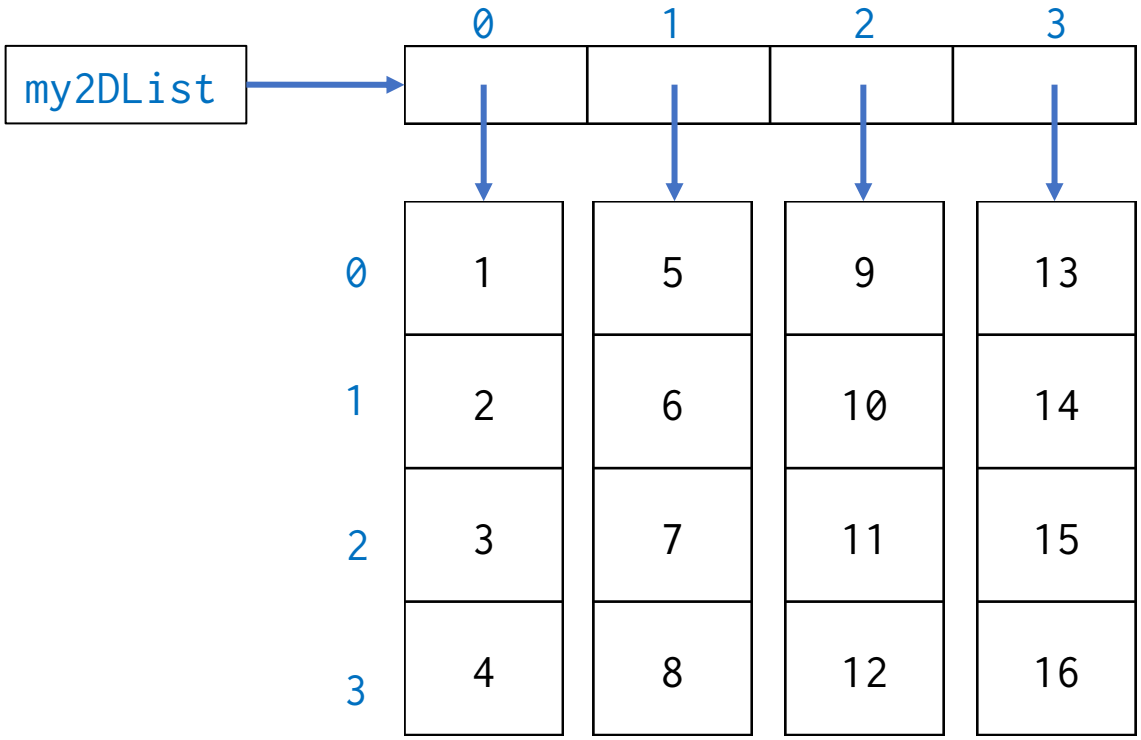
```
>>> for (key,value) in myDict.items():
...     print(key,' ==> ',value)
...
('a', ' ==> ', '1')
('c', ' ==> ', '3')
('b', ' ==> ', '2')
```

Nested Loops

- Loops can be nested just like the if/else statements.
- Indentation is again the key to creating nested loops.
- In a 2 level nested loop with x iterations on the outer loop and y iterations in the inner loop:
  - All statements in the outer loop will be executed x times
  - All statements in the inner loop will be executed x*y times

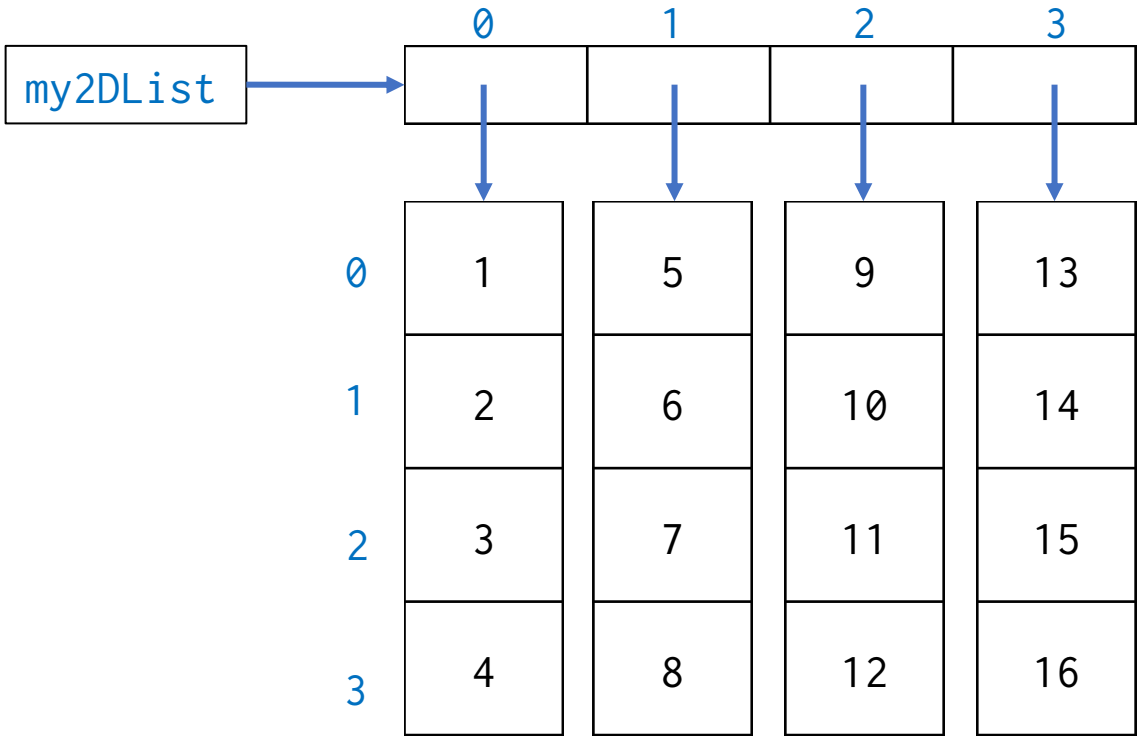
MultiDimensional Lists

```
my2DList = [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]
```



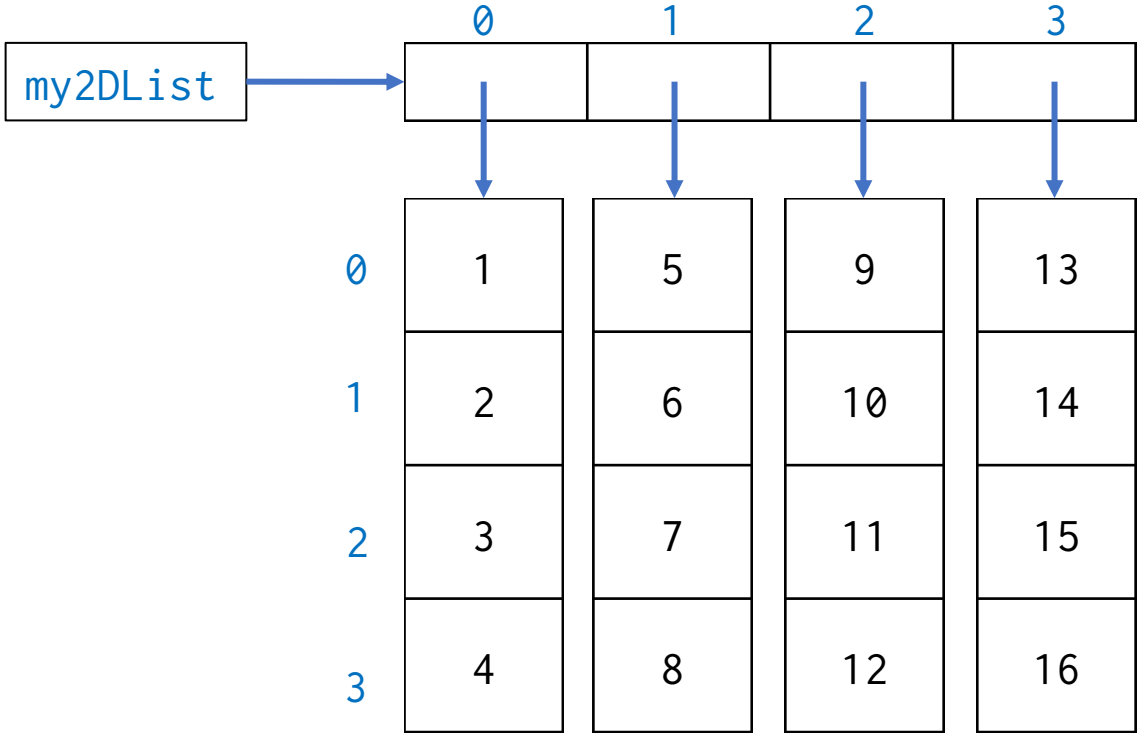
MultiDimensional Lists

```
my2DList = [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]
for x in range(len(my2DList)):
        insideList=my2DList[x]
```



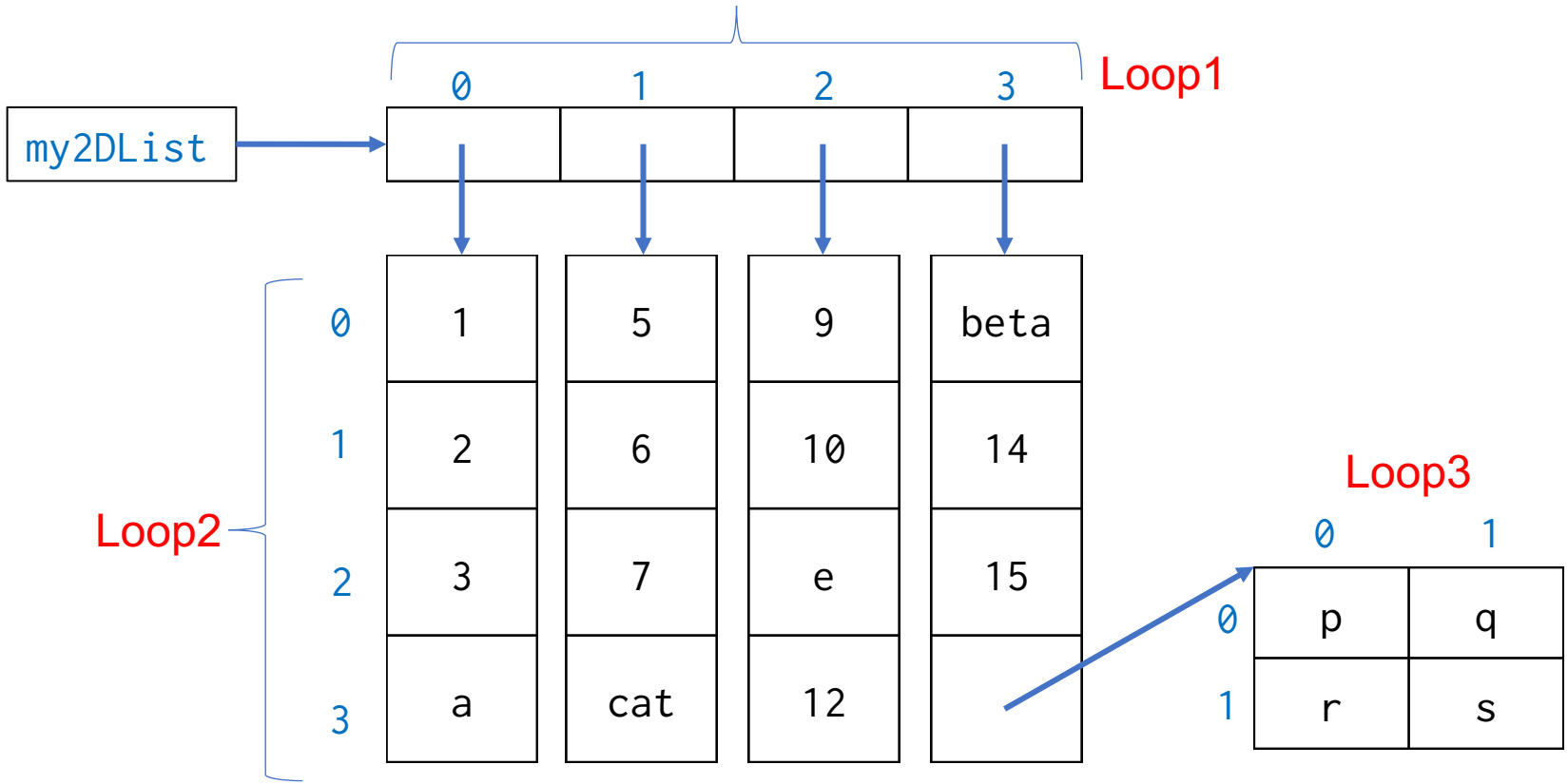
MultiDimensional Lists

```
my2DList = [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]
for x in range(len(my2DList)):
        for y in range(len(my2DList[x])):
                print my2DList[x][y]
```



Arbitrary dimensional Lists

```
subL = [['p','q'],['r','s']]
my2DList = [[1,2,3,'a'],[5,6,7,'cat'],[9,10,'e',12],['beta',14,15,subL]]
```



Summary: Conditions and loops

- Conditional statements with the proper comparison and boolean operators allow the creation of alternate execution paths in the code.
- Loops allow repeated execution of the same set of statements on all the objects within a sequence.
- Using an index based for loop is best suited for making changes to items within a list.
- Always ensure that your exit condition will be met.