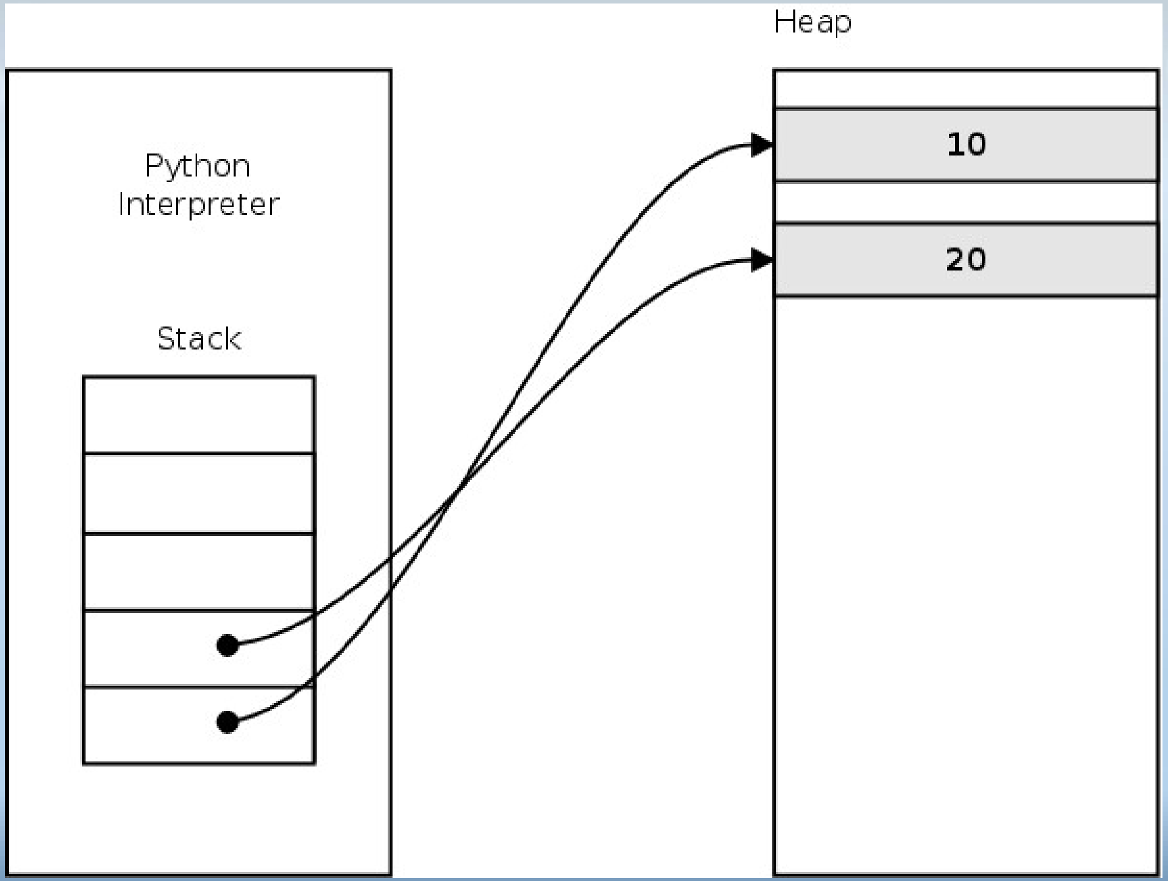
Python’s Stack Machine



Python memory manager

- **Object Storage**: In Python, every value is an object, and objects are allocated memory space dynamically as needed. Objects can be of various types, such as integers, floats, strings, lists, dictionaries, and custom classes. Each object has a header containing information about the object's type and other details, followed by the memory space allocated for the object's data.
- **Reference Counting**: Python uses a technique called reference counting to keep track of how many references (pointers) exist to an object. Each object has a reference count, which is incremented when a new reference is created and decremented when a reference is deleted. When the reference count reaches zero, indicating no references to the object, the memory occupied by the object is deallocated.
- **Garbage Collection**: In addition to reference counting, Python also employs a garbage collector to detect and deallocate objects that have cyclic references or are no longer reachable. The garbage collector periodically identifies and collects such objects, freeing up the memory they occupied.
- **Memory Optimization**: Python's memory manager employs several techniques to optimize memory usage. These include interning small integers and some string literals, reusing memory for small objects, and utilizing memory pools for frequently used object types.
- **Memory Management Models**: Python provides different memory management models. The standard model, called CPython, uses a combination of reference counting and garbage collection. Other Python implementations, such as Jython and IronPython, may use different memory management approaches based on the underlying runtime environment.

Allocation of variables in memory

- For the **built-in numeric data types**. When you assign the value of a to b using the statement `b = a`, it creates a new reference to the value of a at that point in time. After the assignment, b and a are independent of each other.

Therefore, in the code `a = 3; b = a; a = 5`, when you print b, it will give the value 3 because b still holds the original value of a before the reassignment.

- For **sequence types**. Python creates a dynamic link between a and b

```
a = [3]   # Create a list
b = a     # Assign the reference of `a` to `b`
a[0] = 5  # Update the value of the first element of `a`
print(a)  # Output: [5]
print(b)  # Output: [5] not [3]
```

To avoid b referencing the same list as a

- Using the `copy()` method (shallow copy): `b = a.copy()`
- Using slicing (shallow copy): `b = a[:]`
- Using the `deepcopy()` method from copy module (deep copy): `b = a. deepcopy()`

Difference between shallow and deep copy

- copy() method

```
a = [[1, 2], [3, 4]]
b = a.copy()  # Create a copy of `a`
# Modify the nested list in `a`
a[0][0] = 5

print(a)  # Output: [[5, 2], [3, 4]]
print(b)  # Output: [[5, 2], [3, 4]]
```

- deepcopy() method

```
import copy

a = [[1, 2], [3, 4]]
# Create a deep copy of `a`
b = copy.deepcopy(a)
# Modify the nested list in `a`
a[0][0] = 5

print(a)  # Output: [[5, 2], [3, 4]]
print(b)  # Output: [[1, 2], [3, 4]]
```

Introduction to Python (*hour 5*)

- Conditional statements (if, else, elif).
- Comparison operators and logical operators.
- Using conditions to make decisions.

Boolean (logical) operations

- **Logical AND** (`and`): The and operator returns True if both operands are `True`, and `False` otherwise. It evaluates the expressions from left to right and stops as soon as a False value is encountered.

```
a = True
b = False
result = a and b # False
```

- **Logical OR** (`or`): The or operator returns True if at least one of the operands is `True`, and `False` if both operands are `False`. It evaluates the expressions from left to right and stops as soon as a `True` value is encountered.

```
a = True
b = False
result = a or b # True
```

- **Logical NOT** (`not`): The not operator negates the Boolean value of its operand. It returns `True` if the operand is `False`, and `False` if the operand is `True`.

```
a = True
result = not a # False
```

Boolean Algebra Truth Tables

A	B	A and B
True	True	True
False	True	False
True	False	False
False	False	False

A	B	A or B
True	True	True
False	True	True
True	False	True
False	False	False

A	not A
True	False
False	True

Boolean comparison operators

- Comparison operators are used to compare values and return Boolean results. Common comparison operators in Python include:
- Equal to (==)
- Not equal to (! =)
- Greater than (>)
- Less than (<)
- Greater than or equal to (>=)
- Less than or equal to (<=)

```
x = 5
y = 10
result1 = x == y # False
result2 = x < y # True
```

Integer comparison operators

Comparison operators are used to compare two integers and return Boolean results.

- Equal to (==)
- Not equal to (!=)
- Greater than (>)
- Less than (<)
- Greater than or equal to (>=)
- Less than or equal to (<=)

```
x = 5
y = 10
result1 = x == y # False
result2 = x < y # True
```

Float comparison operations

Comparison operators are used to compare two floating-point numbers and return Boolean results. Common comparison operators in Python include:

- Equal to (==)
- Not equal to (!=)
- Greater than (>)
- Less than (<)
- Greater than or equal to (>=)
- Less than or equal to (<=)

```
x = 2.5
y = 4.7
result1 = x == y # False
result2 = x < y # True
```

String comparison operations

- Equal to (==) checks if two strings are equal, i.e., they have the same sequence of characters.
- Not equal to (!=) checks if two strings are not equal, i.e., they have different sequences of characters

```
str1 = "Hello“
str2 = "World“
result1 = str1 == str2 # False
result2 = str1 != str2 # True
```

- Greater than (>), Less than (<), Greater than or equal to (>=), Less than or equal to (<=): These comparison operators compare two strings based on their lexicographical order (compares strings character by character (one by one, from left to right), based on the Unicode values of the characters). The comparison stops as soon as a character with different Unicode value is found.

```
str1 = "Apple“
str2 = "Banana“
result1 = str1 < str2 # True
result2 = str1 >= str2 # False
```

The comparison is case-sensitive, meaning uppercase letters are considered to be less than lowercase letters.
If you want to perform case-insensitive string comparison, you can convert the strings to lowercase or uppercase before comparing.

Conditional statements

Conditional statements in Python are used to execute different blocks of code based on certain conditions. The main conditional statements in Python are `if`, `else`, and `elif` (short for "else if"). Here's an explanation of how these statements work:

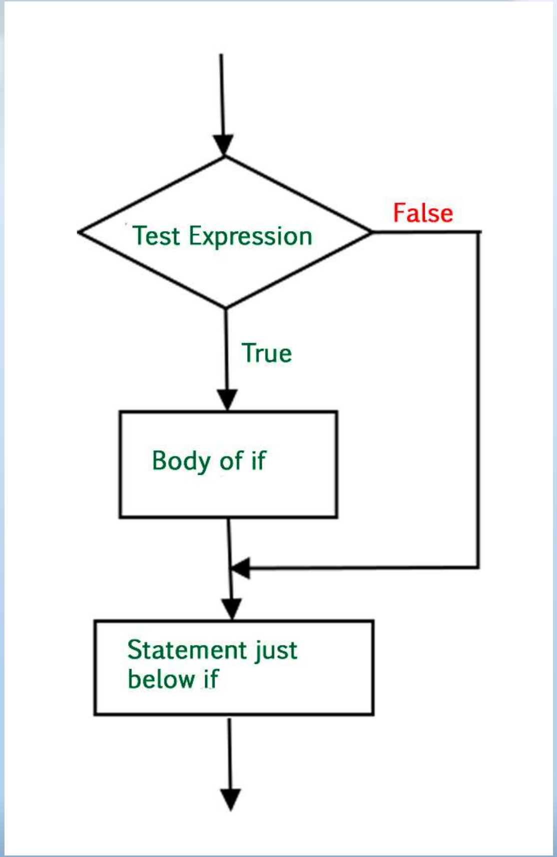
- `if` statement:
  - The if statement allows you to check a condition and execute a block of code only if the condition is true.
  - If the condition evaluates to true, the indented block of code following the if statement is executed.
  - If the condition is false, the block of code is skipped.

```
x = 10
if x > 0:
    print("x is positive")
```

- `else` statement:
  - The else statement is used in conjunction with the if statement to specify an alternate block of code to be executed when the if condition is false.
  - The else block is executed only if the preceding if condition is false.

```
x = -5
if x > 0:
    print("x is positive")
else:
    print("x is non-positive")
```

WARNING! Alignment

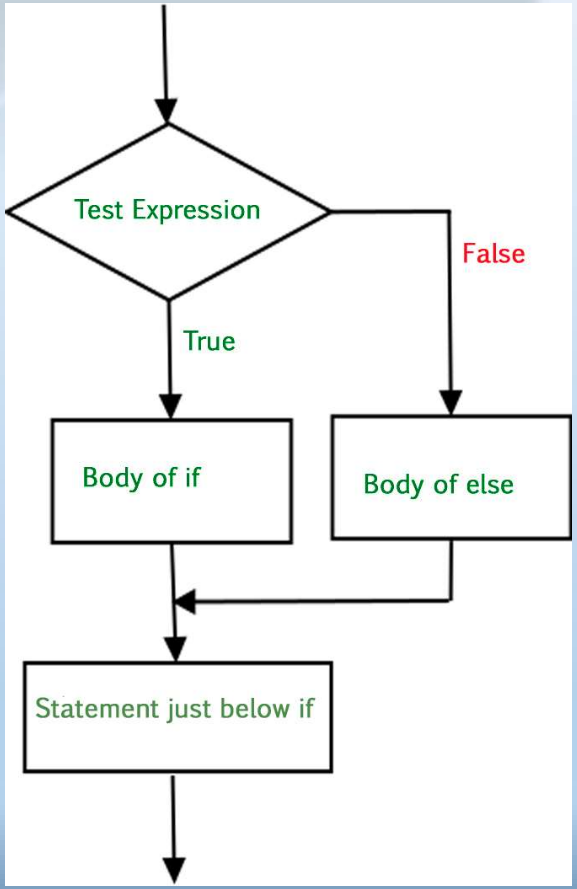


Conditional statements

- `elif` statement:
  - The `elif` statement allows you to check additional conditions after the initial if condition.
  - It is used when you have multiple conditions to check, and you want to specify different code blocks to execute based on these conditions.
  - The `elif` statement is followed by a condition that is evaluated only if the preceding if or `elif` condition(s) are false.
  - You can have multiple `elif` statements, but only the block of code corresponding to the first true condition will be executed. Subsequent conditions are not evaluated.

```
x = 0
if x > 0:
    print("x is positive")
else:
    if x < 0:
        print("x is negative")
    else:
        print("x is zero")
```

```
x = 0
if x > 0:
    print("x is positive")
elif x < 0:
    print("x is negative")
else:
    print("x is zero")
```



Complex statement

To make complex statement (or block of statements):

- Use semicolon “;” to write several statement in one line:

```
a = 10; b = 2; d = 223
```

- Use semicolon text alignment:

```
if False:
     a = 10
     b = 2
     d = 223
print(d) # Error!
```

```
if False:
     a = 10
     b = 2
d = 223
print(d) # 223
```

Using conditions to make decisions

To use conditions to make decisions in Python, you can use conditional statements such as `if`, `else`, and `elif`. These statements allow you to specify different code blocks to be executed based on the outcome of certain conditions. Here's an example to demonstrate how conditions are used to make decisions:

```
# Let's assume we have a variable representing the current temperature
temperature = 28

# We can use an if statement to make a decision based on the temperature value
if temperature > 30:
    print("It's hot outside!")
elif temperature < 15:
    print("It's cold outside!")
else:
    print("The weather is moderate.") # Output: The weather is moderate.
```

By using conditions in your code, you can make decisions and control the flow of your program based on specific conditions or criteria. This allows your program to handle different scenarios and adapt its behavior accordingly.

End of 3-4 hours

Thank you for your attention

Introduction to Python (*hour 5*)

- Introduction to loops (**for** and **while** loops).
- Iterating over sequences and ranges.
- Loop control statements (**break** and **continue**).

Introduction to loops

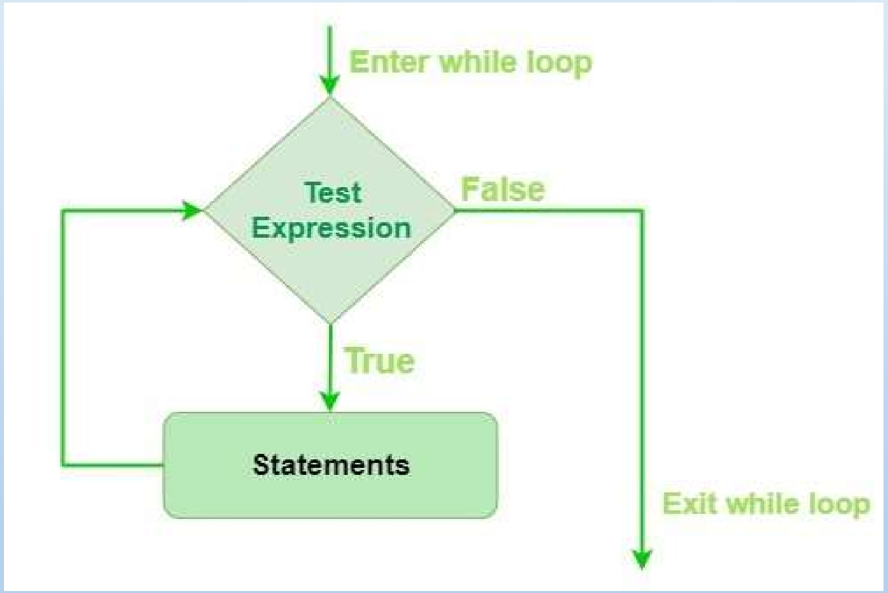
Loops are control structures that allow you to repeatedly execute a block of code.
There are two main types of loops in Python:

- `while` loop
- `for` loops

While Loop

While Loop is used to execute a block of statements repeatedly until a given condition is satisfied. And when the condition becomes false, the line immediately after the loop in the program is executed.

The condition is checked before each iteration. If the condition is true, the indented block of code following the while statement is executed. If the condition is false, the loop is exited, and the program continues with the next line of code.



While examples

- Example 1: Using a while loop to countdown from 5 to 1

```
count = 5
while count > 0:
   print(count, end='')
   count -= 1
# Output: 5 4 3 2 1
```

- Example 2: Using a while loop to prompt the user until valid input is given

```
name = ""
while not name:
    name = input("Enter your name: ")
print("Hello, " + name)
```

For loop

`for` loop is used when you want to iterate over a sequence or a collection of items, such as a list, tuple, string, or range of numbers.

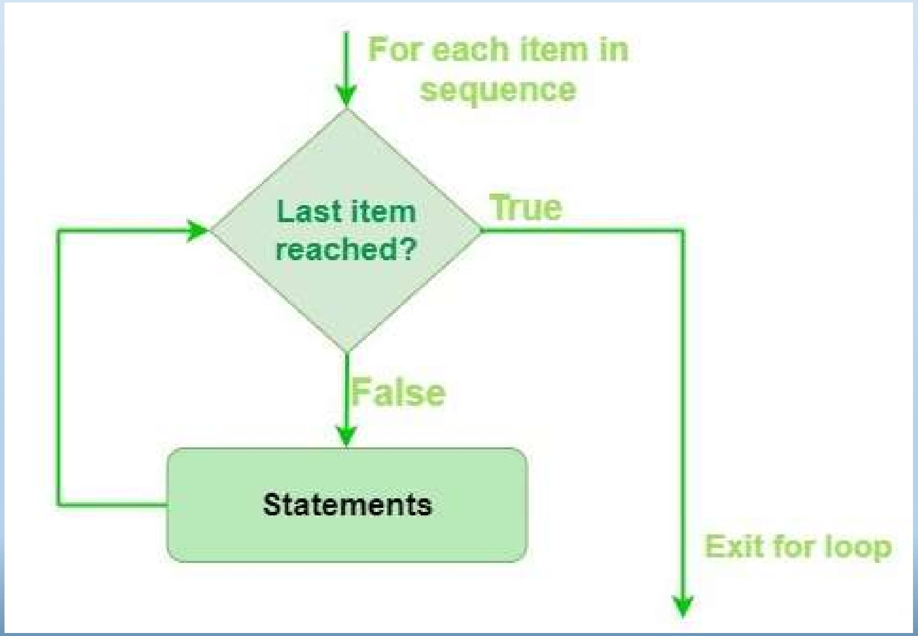
- The loop variable takes on each value in the sequence one by one, and the indented block of code following the for statement is executed for each iteration.
- The loop continues until all the items in the sequence have been processed.

- Example 1: Iterating over a range of numbers

```
for i in range(1, 5):
    print(i, end='')
# Output:  1 2 3 4
```

- Example 2: Iterating over a list

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
# Output:
# apple
# banana
# cherry
```



Iterating over sequences and ranges

In Python, you can use loops to iterate over sequences and ranges. Iterating over a sequence means accessing eac

1.Iterating over a sequence:

- Sequences can be lists, tuples, strings, or any other iterable object.
- The loop variable takes on each value in the sequence one by one, and the indented block of code following the

python

•# Iterating over a list fruits = ["apple", "banana", "cherry"] for fruit in fruits: print(fruit) # Output: # apple # banana # cherry # Iterating over a string message = "Hello, World!" for char in message: print(char) # Outp

•Iterating over a range:

•A range represents a sequence of numbers.

•You can use the range() function to generate a range of numbers.

•The loop variable takes on each value in the range one by one, and the indented block of code following the for statement is e

python

1.# Iterating over a range of numbers for i in range(1, 5): print(i) # Output: # 1 # 2 # 3 # 4 # Iterating over a range with a step value for i in range(1, 10, 2): print(i) # Output: # 1 # 3 # 5 # 7 # 9

By using for loops, you can iterate over sequences like lists, tuples, and strings, as well as ranges of numbers. This allows you to access and process each element or value in the sequence or range individually. It's a convenient way to perform operations on multiple items

Loop control statements

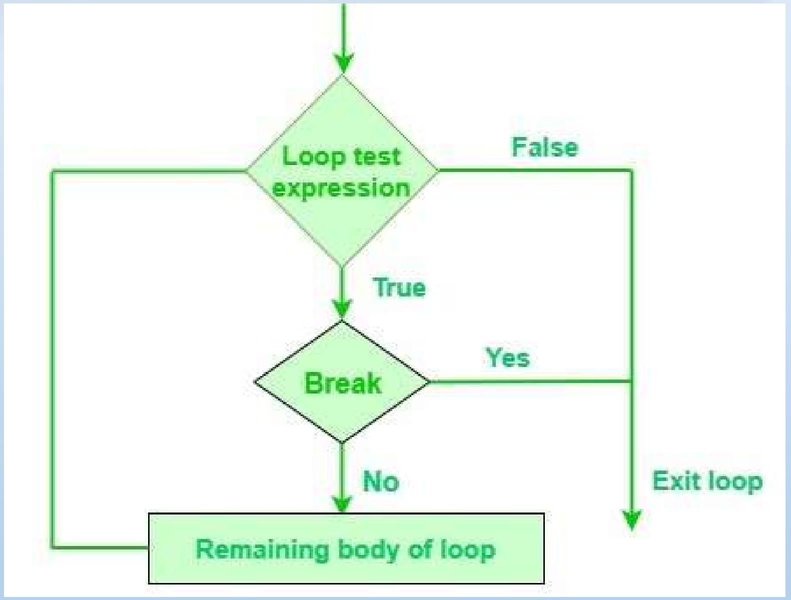
- break statement:

  The break statement is used to exit or terminate the loop prematurely. When encountered inside a loop, the break statement immediately exits the loop, regardless of whether the loop condition is still true or not.

It is often used when a certain condition is met, and you want to stop the loop execution.

```
# Example: Using break to exit a loop
numbers = [1, 2, 3, 4, 5]
for num in numbers:
   if num == 3:
       break print(num)
# Output:
# 1
# 2
```

```
count = 1
while True:
   print(count, end='')
   if count == 3:
       break
   count += 1
# Output: 1 2
```



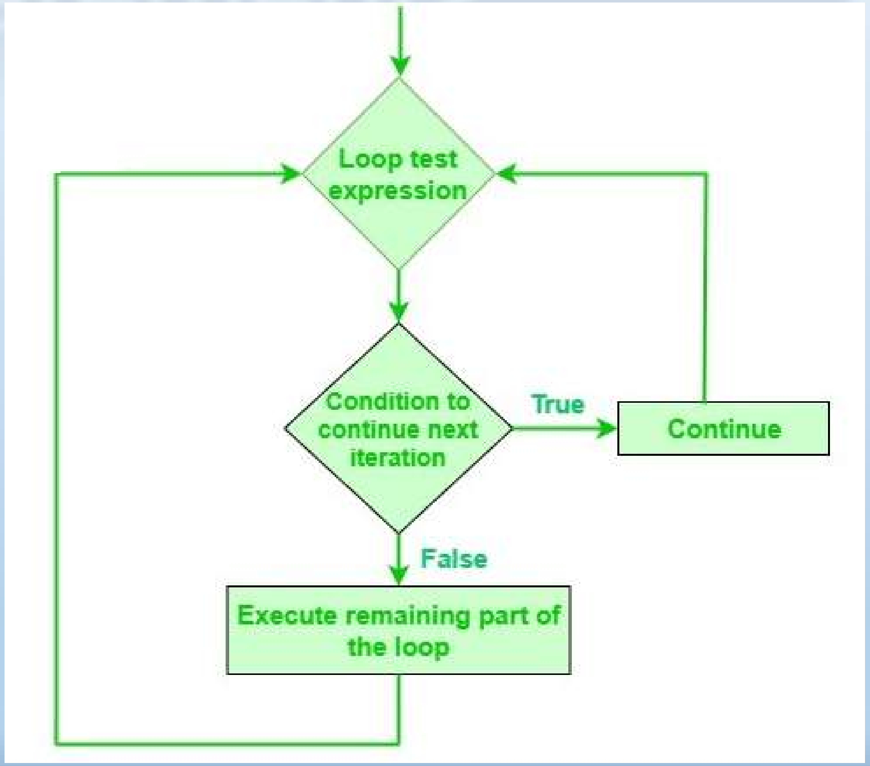
Loop control statements

- `continue` statement:

  The continue statement is used to skip the current iteration and proceed to the next iteration of the loop. When encountered inside a loop, the continue statement immediately jumps to the next iteration, skipping the remaining code block for the current iteration.

It is often used when you want to bypass certain iterations based on a specific condition.

```
# Example: Using continue to skip an iteration
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num % 2 == 0:
        continue
    print(num)
# Output:
# 1
# 3
# 5
```


Loop test expression
Condition to continue next iteration
True
Continue
False
Execute remaining part of the loop

Loop control statements

- `pass` statement:

  The pass statement is a placeholder statement that does nothing. It is used when you need a statement syntactically but don't want to execute any code.

It is often used as a placeholder for code that will be implemented later or as a placeholder inside an empty loop or conditional block.

```
# Example: Using pass as a placeholder
for i in range(5):
    # Code to be implemented later
    pass
# Output: No output, as pass does nothing
```

List Comprehensions

A list comprehension is a concise way to create a new list by performing an operation on each element of an existing **iterable** (such as a list, tuple, or string). It follows the syntax [expression for item in iterable if condition], where:

- expression is the operation or transformation to apply to each item,
- item is a variable representing each item in the **iterable**,
- **iterable** is the source from which the items are taken,
- condition is an optional filter that allows you to include only certain items based on a specified condition.

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = [num ** 2 for num in numbers]
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

Generators

Generators provide an efficient way to create iterators without storing the entire sequence in memory. They are defined using a similar syntax to list comprehensions but enclosed within parentheses (expression for item in iterable if condition). Generators produce elements on-the-fly as requested, making them memory-efficient and suitable for working with large or infinite sequences.

```
def squares(start, end):
    for num in range(start, end + 1):
        yield num ** 2

squared_values = squares(1, 5)
for value in squared_values:
    print(value)
# Output: 1, 4, 9, 16, 25
```

End of 4-5 hours

Thank you for your attention

Introduction to Python (*hour 6*)

- Introduction to functions.
- Defining and calling functions.
- Function parameters and return values.
- Function arguments (positional, keyword, default).
- Variable scope (global and local variables).
- Recursive functions.

Defining functions

```
def fib(n):
  """Print a Fibonacci series up to n."""
  a, b = 0, 1
  while b < n:
    print b,
    a, b = b, a+b

# function usage (function call)
fib(2000)
```

- First line is *docstring*
- first look for variables in local, then global
- need global to assign global variables

Function call

In Python, function calls are used to execute a defined function with the provided arguments (if any). When making a function call, you typically follow the function name with parentheses ().

```
function_name(argument1, argument2,
  ...)
```

- Calling a Function **without Arguments**:

```
def greet():
 print("Hello, World!")
greet() # Output: "Hello, World!"
```

- Calling a Function **with Arguments**

```
def greet(name):
    print(f"Hello, {name}!")
greet("Alice")  # Output: "Hello,
Alice!“
greet("Bob")  # Output: "Hello, Bob!“
```

- Calling a Function and **Capturing the Return Value**

```
def add_numbers(a, b):
    return a + b
result = add_numbers(3, 4)
print(result)  # Output: 7
```

Defining functions

```
def fib(n):
  """Print a Fibonacci series up to n."""
  a, b = 0, 1
  while b < n:
    print b,
    a, b = b, a+b

# function usage (function call)
fib(2000)
```

- First line is *docstring*
- first look for variables in local, then global
- need global to assign global variables

Function arguments

Functions can have different types of arguments:

- **Positional Arguments** are the most basic type of arguments and are passed to a function based on their position. The order in which you provide the arguments matters, as it determines how they are assigned within the function:

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet("Alice", 25)  # Positional arguments
```

- **Keyword Arguments** are passed with a name-value pair, allowing you to specify the argument based on its parameter name rather than position. This flexibility can be useful when dealing with functions that have many arguments or when you want to pass only specific arguments

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet(age=30, name="Bob")  # Keyword arguments
```

Function arguments

- **Mixing Positional and Keyword Arguments.** We can mix positional and keyword arguments in a function call. However, positional arguments must come before keyword arguments:

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet("Alice", age=25)  # Positional arguments
```

- **Default Parameters** allow you to specify a default value for an argument. If no value is provided for the argument during the function call, the default value is used

```
def greet(name, age=30):
    print(f"Hello, {name}! You are {age} years old.")

# Using default parameter value
greet(“Alice")
# Overriding default parameter value
greet("Bob", 35)
```

Operators for Positional Arguments

- * Operator for Positional Arguments (**Unpacking a List**): When * is used in a function definition, it allows you to pass a variable number of positional arguments to the function as a list. Inside the function, these arguments are treated as a single iterable

```
def concatenate(*args):
    result = "“
    for arg in args:
        result += arg
    return result
print(concatenate("I", " ", "am", " ", "Python"))
# Output: "I am Python“
```

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

lst = ["Alice", 25]
greet(*lst)
```

Operators for Positional Arguments

- `**` Operator for Keyword Arguments (Unpacking a Dictionary): The `**` operator allows you to pass a variable number of keyword arguments to a function as a dictionary. Inside the function, these arguments are treated as a dictionary of key-value pairs:

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=25, city="New York")
# Output:
# name: Alice
# age: 25
# city: New York
```

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

dct = [“name”: "Alice", age: 25]
greet(**dct)
```

Operators for Positional Arguments

- We can use `*` and `**` together in a function definition to handle both positional and keyword arguments:

```
def process_data(*args, **kwargs):
    for arg in args:
        print(arg)
    for key, value in kwargs.items():
        print(key, value)

process_data(1, 2, 3, name="Alice", age=25)

# Output:
# 1
# 2
# 3
# name Alice
# age 25
```

Variable scope

- **Global variables** are defined outside of any function or block, making them accessible from anywhere within the code. They have a global scope, meaning they can be accessed both inside and outside functions:

```
# Global variable
global_var = "I am global“

def print_global():
    print(global_var)  # Accessing global variable inside the function

print_global()  # Output: "I am global“
print(global_var)  # Output: "I am global"
```

Variable scope

- **Local variables** are defined within a function or block and have a local scope, meaning they are only accessible within that specific function or block. They are not visible or accessible from outside:

```
def print_local():
    local_var = "I am local“
    print(local_var)  # Accessing local variable inside the function

print_local()  # Output: "I am local“
print(local_var)  # Error: NameError: name 'local_var' is not defined
```

Variable scope

- **Global and Local Variables with the same name**: If a variable with the same name exists both globally and locally within a function, the local variable takes precedence within the function:

```
global_var = "I am global“
def print_var():
    global_var = "I am local“
    print(global_var)  # Accessing local variable inside the function

print_var()  # Output: "I am local“
print(global_var)  # Output: "I am global"
```

Recursive functions

Recursive functions are functions that call themselves within their own definition. They provide a way to solve complex problems by breaking them down into smaller, more manageable subproblems. Recursive functions follow a specific structure:

- **Base Case**: A base case is the condition in which the function does not call itself and instead returns a result directly. It serves as the stopping criterion for the recursion. Without a base case, the recursive function would continue calling itself indefinitely, leading to infinite recursion.
- **Recursive Case**: The recursive case is the part of the function where it calls itself to solve a smaller version of the original problem. By breaking down the problem into smaller subproblems, the function can solve them iteratively until reaching the base case.

Recursive functions (example)

```
def factorial(n):
    # Base case: when n is 0 or 1, return 1
    if n == 0 or n == 1:
        return 1
    # Recursive case: multiply n by the factorial of (n-1)
    else:
        return n * factorial(n-1)

# Calling the recursive functionresult = factorial(5)
print(result)  # Output: 120
```

Docstrings

- Docstrings in Python are used to document functions, classes, and modules. They provide a way to describe what the code does, explain the purpose of the code, document input and output parameters, and provide usage examples.
- Accessing Docstrings: You can access the docstring of a function, class, or module using the `__doc__` attribute.

```
print(my_function.__doc__)  # Output the docstring for my_function
```

Function Docstrings

- Docstrings for functions are defined as strings immediately after the function definition. They are enclosed in triple quotes (''' or """), and the convention is to use triple double quotes (""").

```
def my_function(arg1, arg2):
    ""“
    This is the docstring for my_function.
    It explains what the function does and how to use it.
    Args:
        arg1 (int): The first argument.
        arg2 (str): The second argument.
    Returns:
        str: The result of the function.
    Examples:
        >>> my_function(10, 'Hello')
        'The result is: HelloHelloHelloHelloHelloHelloHelloHelloHelloHello‘
    ""“
    result = arg2 * arg1
    return f"The result is: {result}"
```

Class Docstrings

- Docstrings for classes are also defined as strings immediately after the class definition. They follow the same convention of using triple double quotes (""").

```
class MyClass:
    ""“

    This is the docstring for MyClass.
    It provides an overview of the class and its purpose.
    ""“

    # Class code here
```

Writing Good Docstrings

To write effective docstrings, follow these guidelines:

- Begin with a summary line that succinctly describes the purpose of the code.
- Use sections to provide more detailed explanations, such as parameters, return values, and examples.
- Use descriptive and clear language to explain the code's behavior.
- Follow the appropriate style guide (e.g., PEP 257) for docstring formatting.

PEP

- PEP stands for "Python Enhancement Proposal." It is a design document that provides information, describes new features, suggests improvements, or proposes changes to the Python programming language or its ecosystem. PEPs are intended to be a means of communication and collaboration among Python developers, allowing them to discuss and document various aspects of the language.
- PEPs cover a wide range of topics, including language syntax, standard library modules, development processes, guidelines, and more. They can be submitted by anyone in the Python community, including developers, users, or enthusiasts, to propose ideas or changes to Python.
- PEPs go through a review and discussion process within the Python community, involving feedback and input from developers. Accepted PEPs may be implemented in future versions of Python, influencing the language's evolution and development.
- PEPs are numbered and categorized based on their type and status. The PEP index (https://peps.python.org/) provides a comprehensive list of all PEPs along with their descriptions and statuses. It serves as a valuable resource for understanding Python's design decisions, proposed enhancements, and best practices.

PEP 257

PEP 257 is the Python Enhancement Proposal that provides guidelines for writing docstrings in Python. It covers the recommended conventions and style for documenting modules, functions, classes, and methods.

- Here are some key points from PEP 257:
- Docstring Format:
- Use triple double quotes (""") for multi-line docstrings and single double quote (") for one-line docstrings.
- Include a summary line that starts with a capital letter and ends with a period. It should be a concise description of the object's purpose.
- Separate the summary line from the rest of the docstring with an empty line.
- Use complete sentences for the explanations and descriptions.
- Sections in Docstrings:
- Start with a summary line and an empty line, followed by additional sections if needed.
- Include sections like "Args", "Returns", "Raises", "Examples", etc., as appropriate for the object being documented.
- Use the section headers as subheadings and describe each section accordingly.
- Argument Documentation:
- Document the arguments and their types in the "Args" section.
- Use the :param argname: syntax to specify the argument name.
- Mention the type of each argument, if applicable.
- Provide a description of the argument's purpose, expected value, or any special considerations.
- Return Value Documentation:
- Document the return value in the "Returns" section.
- Specify the type of the return value, if applicable.
- Describe what the function or method returns or any special considerations regarding the return value.
- Other Sections:
- Use the "Raises" section to document any exceptions that the function or method can raise.
- Use the "Examples" section to provide usage examples or demonstrate how to use the documented object.
- Adhering to the guidelines outlined in PEP 257 helps maintain consistency and readability in your code's documentation. It makes it easier for others (and yourself) to understand and use your code effectively.

peps.python.org/pep-0257

Sphinx

To generate function documentation in PDF or HTML format for all functions with docstrings, you can use documentation generation tools like Sphinx or pydoc. Here's an overview of how you can accomplish this:

- Install Sphinx: Sphinx is a widely used documentation generation tool in the Python ecosystem. You can install it using pip:

```
pip install sphinx
```

- Initialize Sphinx project: In your project directory, navigate to the command line and run the following command to initialize a Sphinx project:

```
sphinx-quickstart
```

This command will prompt you with a series of questions to configure the project. You can choose the desired options, such as project name, author, version, and documentation format (HTML, PDF, etc.).

Sphinx

- Configure Sphinx: Open the generated conf.py file in a text editor and update the extensions section to include the sphinx.ext.autodoc extension:

```
extensions = [
    'sphinx.ext.autodoc',
    # Other extensions
]
```

- Generate documentation: Create a new .rst file in the source directory (e.g., functions.rst) and include the following content:

```
Functions
=========


.. automodule:: module_name
   :members:
   :undoc-members:
```

Replace module_name with the name of the Python module containing the functions you want to document.

Sphinx

- Build documentation: Open the command line and navigate to the project directory. Run the following command to generate the documentation:

```
make html
```

This command will generate HTML documentation based on the Sphinx configuration.

- Export to PDF (optional): If you want to generate a PDF version of the documentation, you can use a tool like PrinceXML or wkhtmltopdf. These tools allow you to convert the HTML output generated by Sphinx to PDF format.

- Alternatively, you can use the sphinx-build command with the latexpdf option to directly generate the documentation in PDF format:

```
sphinx-build -b latex . build/latex
cd build/latex
make all-pdf
```

This will generate a PDF version of the documentation.

End of 5-6 hours

Thank you for your attention