Introduction to Python Programming: Lecture Notes for Freshman Engineering Students

This lecture provides an in-depth introduction to Python programming, tailored for freshman engineering students. It covers essential programming concepts with detailed explanations, examples, and questions inspired by the provided exam documents. The goal is to equip students with a solid foundation in Python, emphasizing how concepts work at a deeper level, while keeping explanations accessible. The lecture is designed for a 90-minute session.

---

1. What is Computation?

Computation is the process of performing calculations or transformations on data to produce results, typically executed by a computer. At its core, computation involves:

- **Input**: Data provided to the program (e.g., user input, files).
- **Processing**: Manipulating data using algorithms or instructions.
- **Output**: Producing results (e.g., printing to the console, storing data).

Python is a **high-level programming language**, meaning it abstracts low-level details (like memory management) and uses human-readable syntax. In contrast, **low-level languages** (e.g., Assembly) are closer to machine code, offering more control but requiring more effort. **Formal languages** are precise, unambiguous languages (like Python) used to instruct computers, unlike natural languages, which are ambiguous.

Example

```
# Input: Get a number from the user
num = int(input("Enter a number: "))
# Processing: Square the number
result = num * num
# Output: Display the result
print("Square is:", result)
```

Question

What is the output of the following code if the user enters 4?

```
num = int(input("Enter a number: "))
print(num * 2)
```

Answer: `8` (The input `4` is multiplied by 2, producing `8`).

---

2. Variables and Types

-> keywords = https://www.w3schools.com/python/python_ref_keywords.asp

Variables are named storage locations in memory that hold data. Python is **dynamically typed**, meaning you don't explicitly declare a variable's type—it's inferred at runtime. Common data types include:

- **int**: Whole numbers (e.g., `5`, `-3`).
- **float**: Decimal numbers (e.g., `3.14`, `-0.5`).
- **str**: Strings (e.g., `"hello"`, `'Python'`).
- **bool**: True or False.
- **list**: Ordered, mutable collection (e.g., `[1, 2, 3]`).
- **tuple**: Ordered, immutable collection (e.g., `(1, 2, 3)`).
- **dict**: Key-value pairs (e.g., `{"name": "Alice", "age": 20}`).

How It Works

When you assign a value to a variable (e.g., `x = 5`), Python allocates memory for the value `5` and associates the name `x` with that memory address. The type is determined automatically based on the value.

Example

```
x = 10        # int
y = 3.14      # float
name = "Alice" # str
is_student = True # bool
print(type(x), type(y), type(name), type(is_student))
```

Question (Inspired by Exam)

Which data structure is ordered, unchangeable, and allows duplicates?
a. Dictionary
b. List

c. Tuple
d. Set
Answer: c. Tuple (Tuples are ordered, immutable, and allow duplicate elements).

---

3. Conditionals and Truth Values

- Order of precedence
  https://rahulrajpvr7d.medium.com/mastering-python-operator-precedence-the-ultimate-shortcut-guide-b0d54e8699d7

Conditionals (`if`, `elif`, `else`) control the flow of execution based on conditions. Conditions evaluate to **boolean** values (`True` or `False`). Common operators include:

- Comparison: `==`, `!=`, `<`, `>`, `<=`, `>=`
- Logical: `and`, `or`, `not`
- Membership: `in`, `not in`
- Identity: `is`, `is not`

Error Types

- **SyntaxError**: Incorrect syntax (e.g., missing colon in `if` statement).
- **TypeError**: Operation on incompatible types (e.g., adding a string and integer).
- **NameError**: Using an undefined variable.
- **IndexError**: Accessing an invalid index in a list or tuple.

How It Works

Python evaluates conditions using short-circuit evaluation for `and` and `or`. For example, in `x > 0 and y < 10`, if `x > 0` is `False`, Python skips evaluating `y < 10`.

Example

```
x = 5
if x > 0 and x < 10:
    print("x is between 0 and 10")
else:
    print("x is out of range")
```

Question (Inspired by Exam)

What is the value of the expression: `5 * 3 * 2 > 10 and 4 + 6 == 11`?
Steps:

1. Evaluate `5 * 3 * 2`: `15 * 2 = 30`.
2. Check `30 > 10`: `True`.
3. Evaluate `4 + 6`: `10`.
4. Check `10 == 11`: `False`.
5. Combine: `True and False` = `False`.
   Answer: `False`.

---

4. Loops

Loops allow repetitive execution of code. Python supports:

- **for loop**: Iterates over a sequence (e.g., list, range, string).
- **while loop**: Repeats as long as a condition is `True`.

Common Problems

- **Infinite loops**: Occur if the loop condition never becomes `False` (e.g., `while True:`).
- **Off-by-one errors**: Incorrect range or index boundaries.
- **Dead code**: Code that is never executed due to unreachable conditions.

Example

```
# For loop with range
for i in range(1, 5):  # Iterates from 1 to 4
    print(i, end=" ")  # Output: 1 2 3 4

# While loop
count = 0
while count < 3:
    print(count)
    count += 1  # Output: 0 1 2
```

Question (Inspired by Exam)

What is the output of:

```
for num in range(-2, -5, -1):
    print(num, end=", ")
```

Answer: `-2, -3, -4,` (Iterates from `-2` to `-4` with step `-1`).

---

5. Functions, Scope, Pre/Postconditions

Functions are reusable blocks of code that perform a specific task. They improve modularity and readability.

Scope

- **Local scope**: Variables defined inside a function are only accessible within it.
- **Global scope**: Variables defined outside functions are accessible everywhere (unless shadowed).
- **Dead code**: Code after a `return` statement is unreachable.

Pre/Postconditions

- **Precondition**: Assumptions about inputs (e.g., input must be a positive integer).
- **Postcondition**: Guarantees about outputs (e.g., function returns a sorted list).

Example

```
def square(num):  # Precondition: num is a number
    return num * num  # Postcondition: returns num squared
print(square(4))  # Output: 16
```

Question (Inspired by Exam)

What is wrong with this code?

```
if (True and (3.21 - 3.11 == 0.5 / 5)):
    print('hello')
```

Answer: Floating-point comparison issue. Due to floating-point precision, `3.21 - 3.11` may not exactly equal `0.1` (`0.5 / 5`). Use a small epsilon for comparison or avoid direct equality checks.

---

6. Recursion

visualization -> https://recursion.vercel.app/

#youtube recursion -> https://www.youtube.com/watch?v=IJDJ0kBx2LM
youtube -> https://www.youtube.com/watch?v=ngCos392W4w&t=1005s

Recursion occurs when a function calls itself to solve a smaller instance of the problem. It requires:

- **Base case**: Stops recursion.
- **Recursive case**: Breaks the problem into smaller subproblems.

Example

```
def factorial(n):
    if n == 0:  # Base case
        return 1
    return n * factorial(n - 1)  # Recursive case
print(factorial(5))  # Output: 120
```

Question (Inspired by Exam)

Write a recursive function to compute the sum of integers from 1 to n.

```
def sum_to_n(n):
    if n <= 1:  # Base case
        return n
    return n + sum_to_n(n - 1)
```

Answer: For `n = 4`, output is `10` (1 + 2 + 3 + 4).

---

7. Strings, Traversing, Slicing

Strings are immutable sequences of characters. You can:

- **Traverse**: Iterate over characters using a loop.
- **Slice**: Extract substrings using `[start:end:step]`.

Example

```
text = "Python"
```

```
# Traversing
for char in text:
    print(char, end=" ")  # Output: P y t h o n
# Slicing
print(text[1:4])  # Output: yth
```

Question (Inspired by Exam)

What is the output of:

```
str1 = "PYTHON programming"
print(str1[2] * str1.index("O"))
```

Answer: `TTTT` (`str1[2] = 'T'`, `str1.index("O") = 4`, so `'T' * 4 = 'TTTT'`).

---

8. Lists, Tuples, Dictionaries

-> geeks for geeks : https://www.geeksforgeeks.org/python/ordered-vs-unordered-in-python/

- **Lists**: Ordered, mutable, allow duplicates (`[1, 2, 3]`).
- **Tuples**: Ordered, immutable, allow duplicates (`(1, 2, 3)`).
- **Dictionaries**: Key-value pairs, mutable, keys are unique (`{"a": 1, "b": 2}`).
- **in operator**: Checks membership (e.g., `3 in [1, 2, 3]`).
- **is operator**: Checks identity (e.g., `x is None`).

Mutability

- **Mutable**: Lists, dictionaries (can be modified in place).
- **Immutable**: Strings, tuples (cannot be changed after creation).
- **Shallow copy**: Copies references to nested objects (`copy.copy()`).
- **Deep copy**: Copies all nested objects recursively (`copy.deepcopy()`).

Example

```
lst = [1, 2, 3]
lst.append(4)  # Modifies list: [1, 2, 3, 4]
tup = (1, 2, 3)  # Cannot modify
d = {"a": 1}
d["b"] = 2  # Modifies dict: {"a": 1, "b": 2}
```

Question (Inspired by Exam)

What is the output of:

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(my_list[5::-2])
```

Answer: `[6, 4, 2]` (Slice from index 5 backward with step -2).

---

9. Classes and Object-Oriented Programming (OOP)

OOP organizes code using **classes** (blueprints) and **objects** (instances). Key concepts:

- **Attributes**: Data stored in an object.
- **Methods**: Functions defined in a class.
- **Inheritance**: A class can inherit attributes/methods from another.

Example

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        return f"Hi, I'm {self.name}"
s = Student("Alice", 20)
print(s.greet())  # Output: Hi, I'm Alice
```

Question

Write a class `Rectangle` with attributes `width` and `height`, and a method `area` to compute the area.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height
```

```
r = Rectangle(5, 3)
print(r.area())  # Output: 15
```

---

10. Flowcharts

A flowchart visually represents program flow using shapes:

- **Oval**: Start/End.
- **Rectangle**: Process (e.g., computation).
- **Diamond**: Decision (e.g., condition).
- **Arrow**: Flow direction.

Example Flowchart (for `if x > 0: print("Positive")`)

- Start (Oval) → Check `x > 0` (Diamond) → If True, Print "Positive" (Rectangle) → End (Oval).
- If False, skip to End.

Question

Draw a flowchart for a loop that prints numbers from 1 to 5.
Answer:

- Start (Oval) → Initialize `i = 1` (Rectangle) → Check `i <= 5` (Diamond) → If True, Print `i` (Rectangle) → Increment `i` (Rectangle) → Back to Check → If False, End (Oval).

Sample Programming Exercise

Write a Python function `is_palindrome(n)` that checks if a number is a palindrome without casting to a string (inspired by the exam).

```
def is_palindrome(n):
    temp = n
    rev = 0
    while n > 0:
        dig = n % 10
        rev = rev * 10 + dig
        n = n // 10
    return temp == rev
```

```
print(is_palindrome(1001))  # Output: True
```
