COL1000
Introduction to Programming

Priyanka Golia

Most (if not all) of the content is borrowed from Prof. Subodh Kumar's slides

Announcements

- Many students are still informing via email about missed labs -- refer to Prof. Subodh Kumar’s mail and use the form link to submit the medical certificate.
- A medical certificate is required only for missed labs. The policy for missed labs (formative assessments/exams) has already been shared via email.
- No medical leave for lectures. Students must maintain 75% attendance; the remaining 25% is for medical or other leaves.
- From this Friday, lab exams will start. Evening sessions are available for support, but attendance has been very low (only 1–2 students).

Nested while loop

Given an integer n and print the multiplication tables for all numbers from 2 to n.

```
1 n = int(input("enter a number"))
2 while n > 1:
3     i = 1
4     print(f"{n}'s table")
5     while i <= 5:
6         print(f"{n} x {i} = {n*i}")
7         i += 1
8     n -= 1
```

Outer loop

Inner loop

Inner loop controlling variable

Outer loop controlling variable

All loops must terminate

```
enter a number3
3's table
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
2's table
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
```

Recall — f-string (f"...") allows you to directly embed variables inside curly braces {}.

For vs While loops

Aspect	`for` loop	while loop
When to use	When the number of iterations is known in advance. Also, called us "**Definite**" loops	When repeating until a condition becomes false. Also, called us "**Indefinite**" loops
Control variable	Automatically handled by Python (ex. `range`)	Must be defined and updated manually
Condition check	Implicitly checked every time through (ex. `range`)	Explicitly written in the `while` condition
Risk of infinite loop	low	High if the condition never becomes false or the variable is not updated
Example use case	Printing numbers 1 to 10, iterating over a list	Keep asking user for input until valid, sum until zero is entered
Syntax emphasis	for i in range(...):	while <condition>:

Recognize Common Errors

- Syntax Error `'hello' = False` `four = twice two`
- NameError `five = two + three` without earlier giving names two or three to any object
- ValueError `int('hello')`
- TypeError `round('a')` `'hello' - 'priyanka'`
- IndexError

```
words = []
words[0] = 3
```

Lists are "mutable," i.e., an object itself may be changed (unlike strings). But not in this way — To modify i^{th} member, there must be one. (X.append(m) will add m to list named X)

```
words.append(10)
words[0] = 3
```

Effectively, word[0] becomes a way to refer to a new value:

- ZeroDivisionError `2.0/0`

Review Concepts

- *for* and *while* loops repeatedly execute the *body*: sequence of statements
  - ➡ Until a termination condition is met
  - ➡ The state changes at the beginning of each iteration of the loop, so it may do new things

Loop control variable Initialized and update implicitly in for loop, explicitly in while loop

- Condition is listed at the entry of the loop, but *break* can quit in the middle
- The statement after loop (if any) always executes after the loop terminates
  - ➡ Ensure that the condition of while loop will eventually become `False`
- Use them to enumerate, e.g., process each element of a collection
- Use them to make gradual progress towards the solution, with each iteration advancing the state of solution closer to the goal

Objects, Variables, References

- Recall every objects has:
  - Type (eg. int, list, str, float, bool, range, etc)
  - Value (the data inside)
  - Id/ Identity (like an address in memory, check with built-in function id())

```
x = 42
print(id(x))      # identity
print(type(x))    # <class 'int'>
print(x)          # value
```

Variables is just a name in your program.
The object lives in memory; the variable is only a **label** pointing to it.

The **link** (arrow) between a variable and an object is a **reference**.

An object always takes up space.

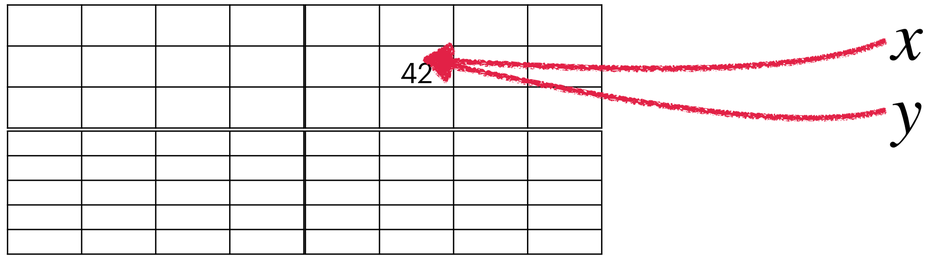
Objects, Variables, References

- Recall every objects has:
  - Type (eg. int, list, str, float, bool, range, etc)
  - Value (the data inside)
  - Id/ Identity (like an address in memory, check with built-in function id())

```
x = 42
print(id(x))      # identity
print(type(x))    # <class 'int'>
print(x)          # value
```

Variables is just a name in your program.
The object lives in memory; the variable is only a **label** pointing to it.

The **link** (arrow) between a variable and an object is a **reference**.



Memory

An object always takes up space.

Multiple variables/names can refer to same object

```
x = 42
y = x
print(id(x), id(y))      # same  identity
>>> 453663663,  453663663
```

Objects, Variables, References

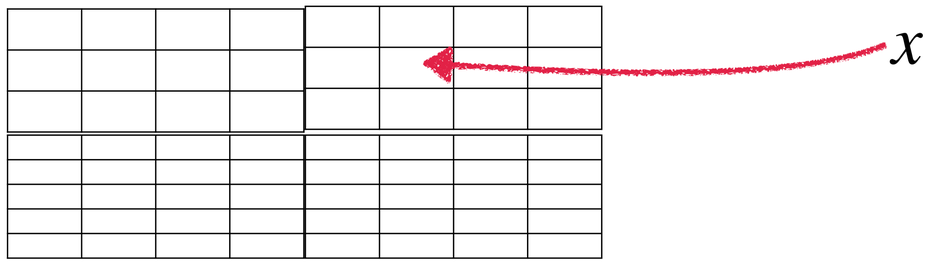
- Recall every objects has:
  - Type (eg. int, list, str, float, bool, range, etc)
  - Value (the data inside)
  - Id/ Identity (like an address in memory, check with built-in function id())

```
x = None
```

None is a special build-in object. It represents “nothing”, “no value”.
Type is “NoneType”, it has an id (memory), but value is empty.

```
x = None    # means "x has no value yet"
```

As a default placeholder.



Memory

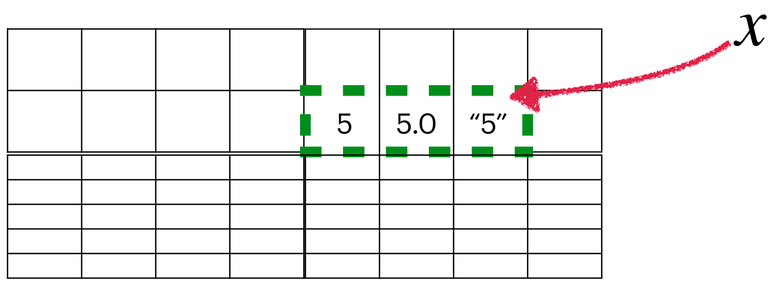
Objects, Variables, References

- Recall every objects has:
  - Type (eg. int, list, str, float, bool, range, etc)
  - Value (the data inside)
  - Id/ Identity (like an address in memory, check with built-in function id())
-

```
x = [5,5.0, "5"]
```

Variables is just a name in your program.
The object lives in memory; the variable is only a **label** pointing to it.

The **link** (arrow) between a variable and an object is a **reference**.



Memory

Mutability vs Immutability

Mutable objects: their contents (value) can be changed in place without changing their identity (id stays the same).

Examples: list, dict, set.

Immutable objects: once created, their values cannot be changed; any “modification” creates a new object with a new identity.

Examples: int, float, bool, str, tuple.