COL1000
Introduction to Programming

Priyanka Golia

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

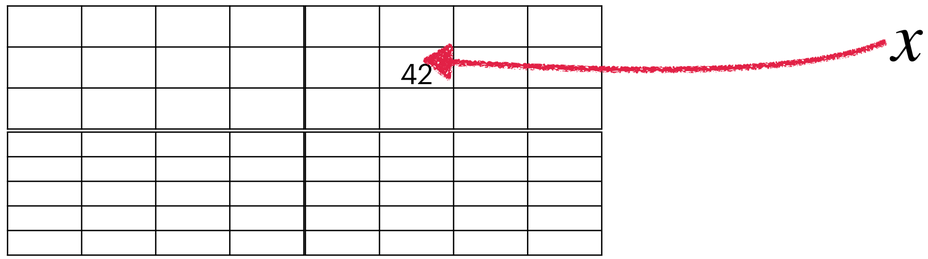
Objects, Variables, References

- 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.

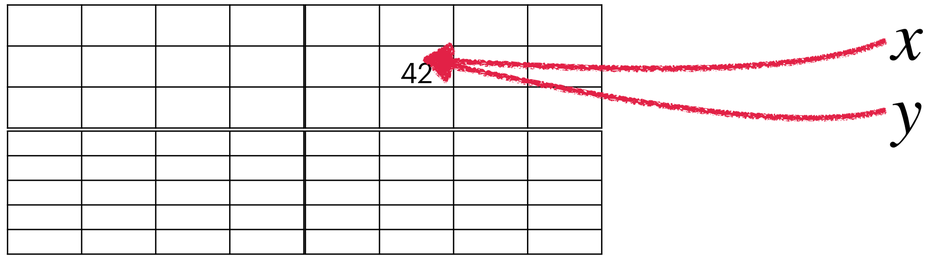
Objects, Variables, References

- 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
```

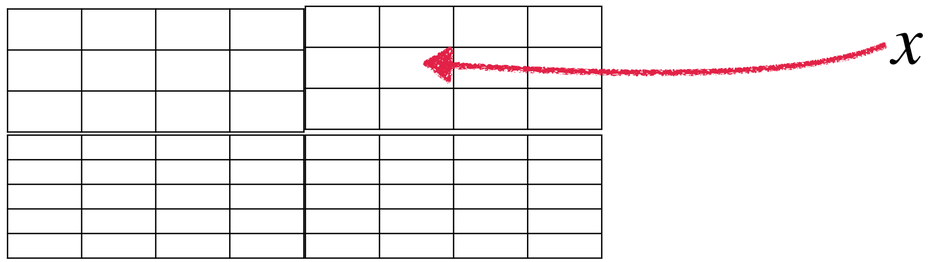
Objects, Variables, References

- 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.

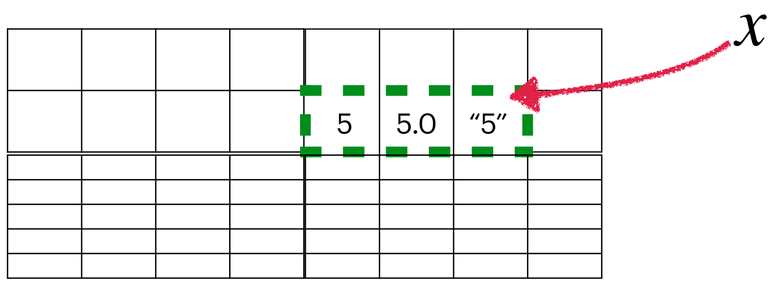
Objects, Variables, References

- 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.

Mutability vs Immutability

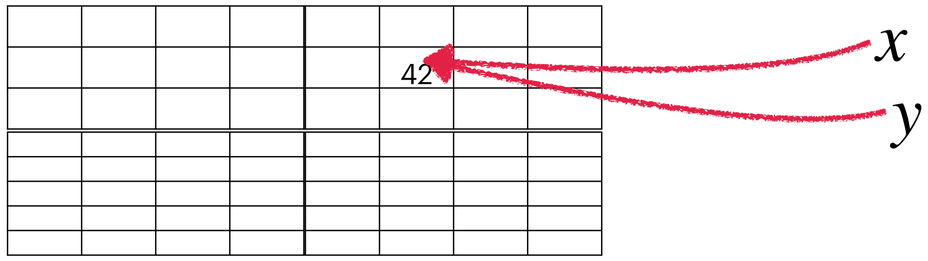
Immutable objects: once created, their values cannot be changed; any “modification” creates a new object with a new identity.

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

```
1 x = 42
2 y = x
3 print("before modification", id(x), id(y))
4 x = x + 1
5 print("after modification", id(x), id(y))
```

Reference copied.

New object created.



Mutability vs Immutability

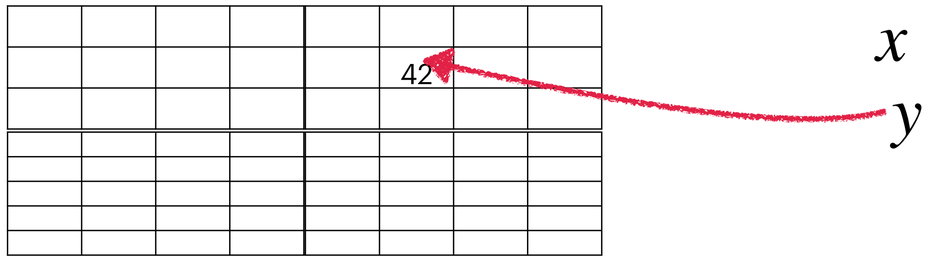
Immutable objects: once created, their values cannot be changed; any “modification” creates a new object with a new identity.

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

```
1 x = 42
2 y = x
3 print("before modification", id(x), id(y))
4 x = x + 1
5 print("after modification", id(x), id(y))
```

Reference copied.

New object created.



Mutability vs Immutability

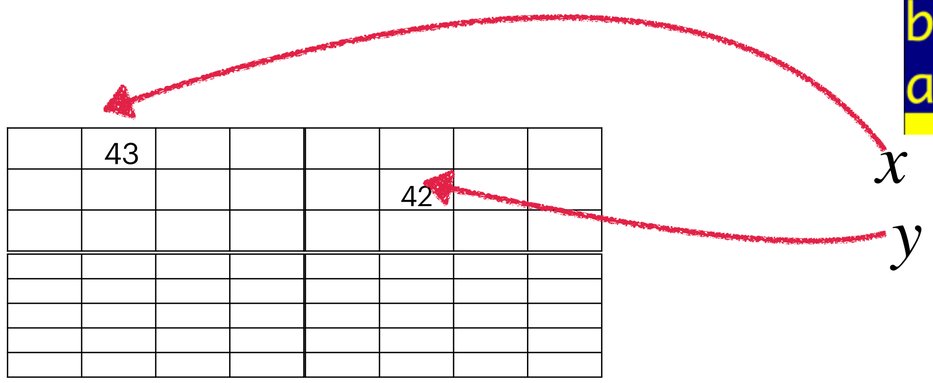
Immutable objects: once created, their values cannot be changed; any “modification” creates a new object with a new identity.

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

```
1 x = 42
2 y = x
3 print("before modification", id(x), id(y))
4 x = x + 1
5 print("after modification", id(x), id(y))
```

Reference copied.

New object created.




```
before modification 9790272 9790272
after modification 9790304 9790272
```

Notice the Id for y remain the same

Mutability vs Immutability

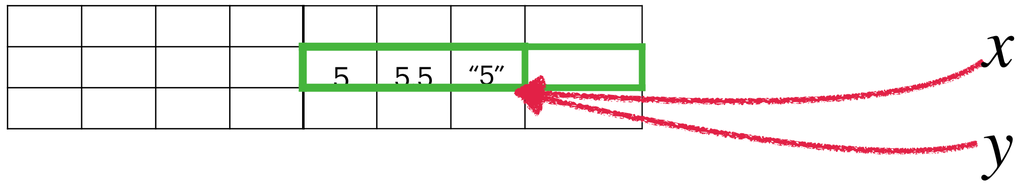
Mutable objects: their contents (value) can be changed in place without changing their identity (id stays the same).

Examples: list, dict, set.

```
1 x = [5,5.5,"5"]
2 y = x
3 print("before modifications, ids", id(x),id(y))
4 print("before modifications:", x,y)
5 x.append("hello")
6 print("after modifications, ids", id(x),id(y))
7 print("after modifications", x,y)
```

Reference copied

Mutable object — in place changes.



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.

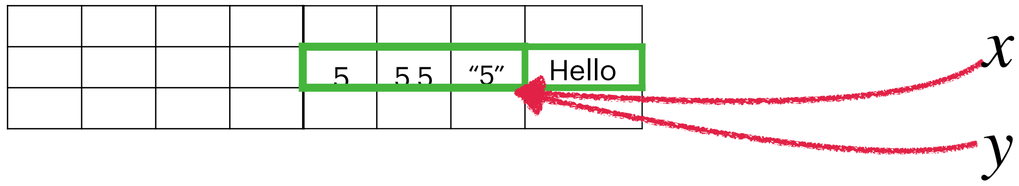
```
1 x = [5,5.5,"5"]
2 y = x
3 print("before modifications, ids", id(x),id(y))
4 print("before modifications:", x,y)
5 x.append("hello")
6 print("after modifications, ids", id(x),id(y))
7 print("after modifications", x,y)
```

Reference copied

Mutable object — in place changes.

```
before modifications, ids 139745809768512 139745809768512
before modifications: [5, 5.5, '5'] [5, 5.5, '5']
after modifications, ids 139745809768512 139745809768512
after modifications [5, 5.5, '5', 'hello'] [5, 5.5, '5', 'hello']
```

Notice that the update in x has been reflected in y.
The IDs of both lists remain the same after the modification.



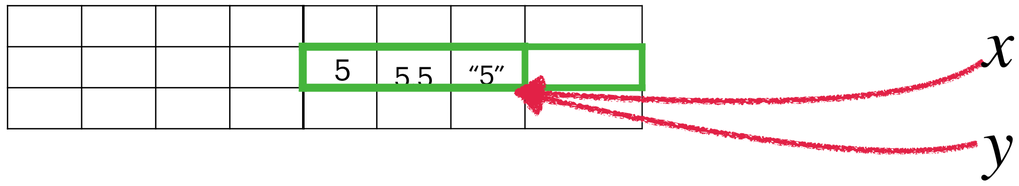
Mutability vs Immutability

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

Examples: list, dict, set.

```
1 x = [5,5.5,"5"]
2 y = x
3 print("before modifications, ids", id(x),id(y))
4 print("before modifications:", x,y)
5 x[0] = 6
6 print("after modifications, ids", id(x),id(y))
7 print("after modifications", x,y)
```

Modification of individual item of item is allowed.



Mutability vs Immutability

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

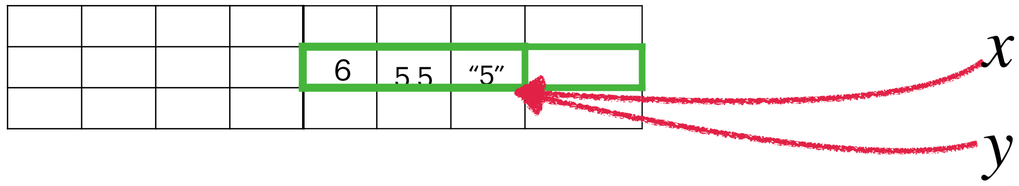
Examples: list, dict, set.

```
1 x = [5,5.5,"5"]
2 y = x
3 print("before modifications, ids", id(x),id(y))
4 print("before modifications:", x,y)
5 x[0] = 6
6 print("after modifications, ids", id(x),id(y))
7 print("after modifications", x,y)
```

Notice that the update in x has been reflected in y.
The IDs of both lists remain the same after the modification.

```
before modifications, ids 140601293828160 140601293828160
before modifications: [5, 5.5, '5'] [5, 5.5, '5']
after modifications, ids 140601293828160 140601293828160
after modifications [6, 5.5, '5'] [6, 5.5, '5']
```

Modification of individual item of item is allowed.



Mutability vs Immutability

```
1 x = [4,5]
2 y = x
3 x.append(6)
4 print(x,y)
5 print(id(x),id(y))
```

```
[4, 5, 6] [4, 5, 6]
139880982917248 139880982917248
```

append(), remove(), etc in-place, no new object

```
1 x = [4,5]
2 y = x
3 x = x + [6]
4 print(x,y)
5 print(id(x),id(y))
```

```
[4, 5, 6] [4, 5]
140151892520000 140151893848128
```

Even for mutable objects, operators like +, -, *, /, // etc create a new object

```
1 x = [4,5]
2 y = x
3 x += [6]
4 print(x,y)
5 print(id(x),id(y))
```

```
[4, 5, 6] [4, 5, 6]
140299113422976 140299113422976
```

Operators like +=, -=, *=, /=, //= . Modify in place for mutable objects, otherwise create new object.

Is vs ==

==

- Value Equality
- Checks whether two objects have the same value.

is

- Identity Equality
- Checks whether two variables refer to the exact same object in memory.
- Equivalent to comparing id(x) == id(y)

Is vs ==

```
1 x = 5
2 y = 5.0
3 print(x == y)
4 print(x is y)
```

```
True
False
```

Value is same, but different objects

```
1 x = [5,6]
2 y = x
3 print(x == y)
4 print(x is y)
5 x.append(7)
6 print(x == y)
7 print(x is y)
8
```


```
1 x = [5, 6]
2 print(x == [5,6])
3 print( x is [5,6]) #object [5,6] will have different id.
4 print( not x is [5,6]) # is has higher precedence than not.
5 print( x is not [5,6])
```

```
True
False
True
True
```

Is vs ==

```
1 x = 5
2 y = 5.0
3 print(x == y)
4 print(x is y)
```

```
True
False
```

Value is same, but different objects

```
1 x = [5,6]
2 y = x
3 print(x == y)
4 print(x is y)
5 x.append(7)
6 print(x == y)
7 print(x is y)
8
```

```
Consol...
True
True
True
True
```

```
1 x = [5, 6]
2 print(x == [5,6])
3 print( x is [5,6]) #object [5,6] will have different id.
4 print( not x is [5,6]) # is has higher precedence than not.
5 print( x is not [5,6])
```

```
True
False
True
True
```

```
Last login: Wed Aug 27 17:48:37
priyanka@Priyankas-MacBook-Pro ~
Python 3.13.1 (main, Dec  3 2024
darwin
Type "help", "copyright", "credi
>>> x = 50
>>> y = 50
>>> print(x is y)
True
>>> x = 300
>>> y = 300
>>> print(x is y)
False
>>> x = 15
>>> y = 15
>>> print(x is y)
True
>>>
```

Avoid using “Is” for immutable objects

May or may not assign at the same Id.

Is vs ==

```
1 x = 5
2 y = 5.0
3 print(x == y)
4 print(x is y)
```

```
True
False
```

Value is same, but different objects

```
1 x = [5,6]
2 y = x
3 print(x == y)
4 print(x is y)
5 x.append(7)
6 print(x == y)
7 print(x is y)
8
```

```
True
True
True
True
```

```
1 x = [5, 6]
2 print(x == [5,6])
3 print( x is [5,6]) #object [5,6] will have different id.
4 print( not x is [5,6]) # is has higher precedence than not.
5 print( x is not [5,6])
```

```
True
False
True
True
```

```
Last login: Wed Aug 27 17:48:37
priyanka@Priyankas-MacBook-Pro ~
Python 3.13.1 (main, Dec  3 2024
darwin
Type "help", "copyright", "credi
>>> x = 50
>>> y = 50
>>> print(x is y)
True
>>> x = 300
>>> y = 300
>>> print(x is y)
False
>>> x = 15
>>> y = 15
>>> print(x is y)
True
>>>
```

Avoid using “Is” for immutable objects

May or may not assign at the same Id.

```
>>> x = [1,3,4]
>>> y = [1,3,4]
>>> print(x is y)
False
>>> x = "hello"
>>> y = "hello"
>>> print(x is y)
True
>>> x = "hello world!"
>>> y = "hello world!"
>>> print(x is y)
False
>>>
```

List is mutable object.
it will always create a new object.

String is immutable object.
Again, may or may not assign same id

Advanced Assignment

Count should match.

$$x, y = value1, value2$$

Assign references from left to right

Both assignments happen **simultaneously** (parallel assignment).

```
x, y = 5, 8
print(x, y) # 5 8
```

This is also useful for **swapping**

```
x, y = 5, 8
print(x, y) # 5 8
x, y = y, x
print(x, y) # 8 5
```

NOT same as

```
x = 5
y = 8
print(x, y) # 5 8
x = y
y = x
print(x, y) # 8 8
```

Sequential assignment

Advanced Assignment

Count should match.

$$x, y = value1, value2$$

$$x, y = 5, 8$$
$$print(x, y) \text{ \# 5 8}$$

Assign references from left to right

```
1 x, y = 10, 15
2 print(x,y)
3 x, y = [10,15]
4 print(x,y)
5 x, y = [10, 15 , 20]
6 print(x,y)
```

unpack list and assign reference.

Count should match.

Two references for three values

```
10 15
10 15
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p10695/lec_main.py", line 3, in <module>
    import lec12
  File "/home/p10695/lec12.py", line 5, in <module>
    x, y = [10, 15 , 20]
ValueError: too many values to unpack (expected 2)
```

Advanced Assignment

Count should match.

$$x, y = value1, value2$$

$$x, y = 5, 8$$
$$print(x, y) \text{ \# 5 8}$$

Assign references from left to right

```
1 x, y = 10, 15
2 print(x,y)
3 x, y = [10,15]
4 print(x,y)
5 x, *y = [10, 15 , 20]
6 print(x,y)
```

```
10 15
10 15
10 [15, 20]
```

First, Right-hand side (RHS) evaluation — The list [10, 15, 20] is created.
Then, Left-hand side (LHS) unpacking — x gets the first element → 10.
*y means: "collect all the remaining elements into a list (or tuple if RHS is tuple)."
So, y becomes [15, 20].

extended iterable unpacking

Advanced Assignment

Count should match.

$$x, y = value1, value2$$

Assign references from left to right

```
1 x, y = 10, 15
2 print(x,y)
3 x, y = [10,15]
4 print(x,y)
5 x, *y = [10, 15 , 20]
6 print(x,y)
```

```
10 15
10 15
10 [15, 20]
```

First, Right-hand side (RHS) evaluation — The list [10, 15, 20] is created.
Then, Left-hand side (LHS) unpacking — x gets the first element → 10.
*y means: "collect all the remaining elements into a list (or tuple if RHS is tuple)."
So, y becomes [15, 20].

extended iterable unpacking

```
x, y = 5, 8
print(x, y) # 5 8
```

More examples

```
x, *y = [1, 2, 3, 4]
# x = 1, y = [2, 3, 4]

*x, y = [1, 2, 3, 4]
# x = [1, 2, 3], y = 4

x, *y, z = [1, 2, 3, 4, 5]
# x = 1, y = [2, 3, 4], z = 5
```

Advanced Assignment

Count should match.

$$x, y = value1, value2$$

Assign references from left to right

More examples

```
x, *y = [1, 2, 3, 4]
# x = 1, y = [2, 3, 4]

*x, y = [1, 2, 3, 4]
# x = [1, 2, 3], y = 4

x, *y, z = [1, 2, 3, 4, 5]
# x = 1, y = [2, 3, 4], z = 5
```

```
x, y = 5, 8
print(x, y) # 5 8
```

Key points about * in unpacking

1. Only **one** starred expression (*variable) is allowed on the left-hand side. (Otherwise Python won't know how to split things.)
2. The starred variable always collects the "leftovers." If there are no leftovers, it becomes an empty list.

```
x, *y = [10]
print(x, y) # 10, []
```

3. Works with any iterable (list, tuple, string, etc.).

```
first, *rest = "hello"
print(first, rest) # 'h', ['e', 'l', 'l', 'o']
```

Review of Data Types

Can query type:

```
type(2.3) != int
```

- Each object is of some type

1	2.3	'2.3'	''	None	True	1_000

  - Literals provide the inherent type of constructed object

[]	[1, 2.3, '2.3']

    - including types of each element of a non-scalar type (aka container)
- References (incl. variables) adopt the type of objects they refer to

```
a = [1]
b = [2, a]
a = [2]
b[1] = 1
```

- An operation only works on objects of types it is designed for

~~list + str~~, ~~str+int~~
~~int in int~~, ~~int>str~~

  - Type of the resulting object depends on the type of operands
  - Functions, including Type conversion (aka type casting), are “operation”

list+list ➜ list
float+int ➜ float
int/ int ➜ float
atype > atype ➜ boo

Types Conversion

Explicit conversion using the type builder function

- int(float_data) truncates towards 0

  int(x) ≠ round(x)
  int(x) ≠ math.floor (x) ☜ towards -∞

- str(most_any_type_of_data) follows in-built rules to convert to string
- int(string_data) or float(string_data) works if the string can be interpreted as number

Implicit conversion is applied on an as-needed basis

- int is converted to float for float operation or operation with float

  because float can (*usually*) represent more numbers than int

- In **`if <condition>:`** the result of condition is converted to bool for testing
- Every type is converted to str for printing

new float created, i remains 3

```
i = 3
f = i/1
i == f
g = i+0.0
print(i,g,f)
```

True

```
3 3.0 3.0
```