

Programming for ML and Data Science

A review of basic programming concepts

Prabhu Ramachandran

Department of Aerospace Engineering, IIT Bombay

2026-02-01

Objects

- Everything in Python is an object (`object`)
- Typically an object
 - Has some state/data
 - Provides methods for functionality
 - Provides interfaces so you can interact with it
- Consider a `list`
 - Has data
 - `lst.append(...)` and other methods
 - Can iterate over it using `for`: programmable behavior

Mutable and immutable

- Two broad categories for all objects
 - Immutable: cannot be changed
 - Mutable: can be changed
- Immutable: numbers, strings, tuples
- Mutable: list, dict, set

Some builtin data types

- Numerical types:
 - Integers: `int`, these are arbitrary precision!
 - Floats: `float`, double precision 64 bits
 - Complex: `1 + 2j`, `complex(1, 2)`

```
1 # Example of large int.  
2 2**(2**8)
```

```
115792089237316195423570985008687907853269984665640564039457584007913129639936
```

```
1 # Complex numbers.  
2 x = complex(2.2, 1.5)  
3 print(abs(x), x.conjugate())
```

```
2.6627053911388696 (2.2-1.5j)
```

- Booleans: `True/False`

Some builtin data types ...

- Sequence types:
 - Lists: `list`, `[]`: mutable
 - Tuples: `tuple`, `()`: immutable
 - Strings: `str`: `'hello'`, `"hello"`, `'''hello'''`,
`"""hello""": immutable`
 - Sets: `set`: behaves like a mathematical set: mutable
 - `range`: `range`: immutable
- Dictionaries: `dict`, `{key: value}`, a mapping (more later)
- Others: `bytes`, `bytearray`, `memoryview` (advanced)

Tuples

- Immutable: `x = (1, 21.2, 'hello')`
 - Cannot change, no append
 - If it contains a list, the list can change

```
1 x = (1, 2, [21, 22])
2 x[-1] = [20, 21] # TypeError!
3 x[-1].append(23) # This will work
```

- Used when returning multiple values from a function
- Tuple expansion

```
1 x, y, z = range(3)
2 x, *y = range(10) # y is a list of remaining elements
3 x, *y, last = range(10)
4 x, y, z = range(10) # ValueError!
```

Operators

- Arithmetic but not always (lists)
- Arithmetic `+`, `-`, `*`, `/`, `**`, `//`, `%` etc.
- `@` is matrix multiplication
- Logical: `not`, `or`, `and`
- Comparison: `<`, `>`, `<=`, `!=`
- Identity: `is`, `is not`
- Containership: `in`, `not in`
- Assignment: `x = 1.0`
 - Augmented assignments: `x += 1.0`
- Bitwise operators: `&`, `|`, `^`, `~`, `<<`, `>>` (and, or, xor, not, left/right shift)
- Don't remember operator precedence, use brackets!!

Floor division

- Produces a float always

```
1 23/2
```

11.5

- Floor division: `//`, same output as left operand

```
1 23//2
```

11

```
1 23.0//2
```

11.0

Identity

```
1 x = 2  
2 y = 1 + 1  
3 x is y
```

True

Containership

```
1 x = ['hello', 'world']
2 'hello' in x
```

True

```
1 x = ('hello', 'world')
2 'hello' in x
```

True

```
1 x = 'hello world'
2 'ello w' in x
```

True

Indexing/Slicing

- Use square brackets: `x[0]`
- Indices start at `0` to `len(x) - 1`
- Slice produces same kind of container:
`x[start:stop:step]`
 - `stop` is not included
 - When not specified `start` is `0`
 - When not specified `stop` is `len(x)`
 - When not specified `step` is `1`
- Dictionaries support non-integer “indices” called **keys**

Looping

- Loops with `for`
- Loops with `while`
- Use `break` to exit the current loop
- Use `continue` to skip execution until end of block

```
1 for i in range(3):  
2     print(i)
```

- While loops look like this:

```
while <conditional>:  
    <block>
```

While example

- Trivial infinite loop:

```
while True:  
    pass
```

- `pass` a syntactic filler to do nothing
- Let us go over this example:

```
1 k = 23  
2 while k != 1:  
3     print(k, end=' ')  
4     if k % 2 == 0:  
5         k //= 2  
6     else:  
7         k = k*3 + 1
```

```
23 70 35 106 53 160 80 40 20 10 5 16 8 4 2
```

Break/continue

- `break` example:

```
1 i = 0
2 while True:
3     if i > 5:
4         break
5     print(i, end=' ')
6     i += 1
```

0 1 2 3 4 5

- `continue` example:

```
1 for i in range(10):
2     if i % 2 == 1:
3         continue
4     print(i, end=' ')
```

0 2 4 6 8

Variables in Python

- Names bound to objects: namespace
- Assignment performs this binding: `x = 1`
- The easy cases (immutable object):

```
1 x = 1
2 y = x
3 del x
4 print(y)
```

1

```
1 x = 1
2 y = x
3 x += 1
4 print(y)
```

1

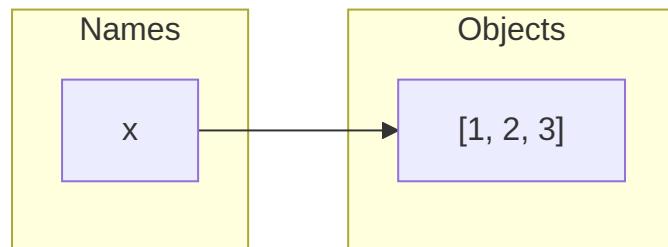
Understanding name binding

- The case of mutable objects

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

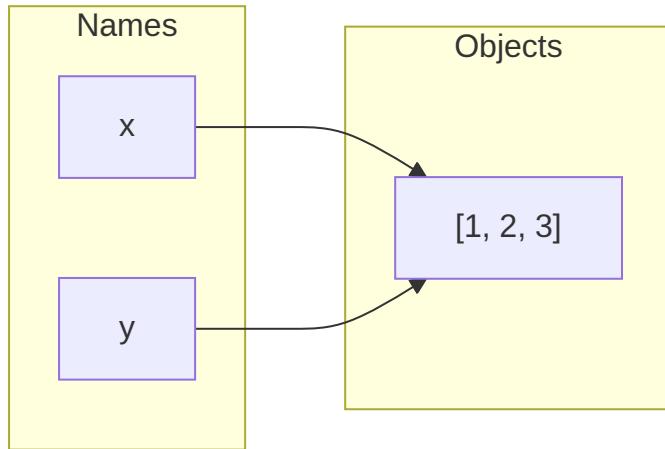
- Let us walk through this:

```
1 x = [1, 2, 3]
```

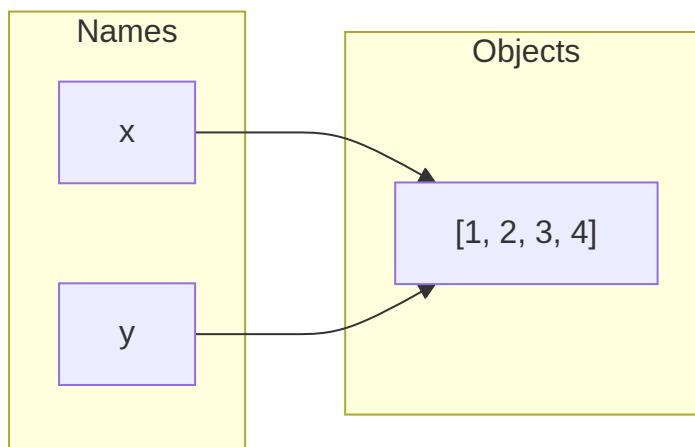


Understanding name binding ...

```
1  y = x
```



```
1  x.append(4)
```



Abstraction with functions

- Functions facilitate reuse
- Functions with no `return` implicitly return `None`

```
1 def func(x, y):  
2     """docstring"""  
3     # ...  
4     return x*y  # Optional
```

- Lambda (anonymous) functions are possible

```
1 func = lambda x, y: x*y
```

Default and keyword arguments

- Note that defaults are evaluated only once

```
1 def greet(name, hi='Hello', repeat=1):  
2     for i in range(repeat):  
3         print(hi, name)  
4  
5 greet("Vinay", "Namaste")  
6 greet(hi="Namaste", name='X! ')
```

Special parameters

- Positional only and keyword only arguments

```
1 def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):  
2     # ...
```

- Examples:

```
1 def normal_arg(arg):  
2     print(arg)  
3  
4 def pos_only_arg(arg, /):  
5     print(arg)  
6  
7 def kwd_only_arg(*, arg):  
8     print(arg)  
9  
10 def combined(pos_only, /, standard, *, kwd_only):  
11     print(pos_only, standard, kwd_only)
```

Some patterns

- Use `None` as a default argument with mutable defaults
- Also to detect if someone passed an argument or not

```
1 def f(x, y=None):  
2     if y is None:  
3         y = []  
4     # ...
```

Functions are “first class” objects

- They can be treated like any other value
- Can be bound to a name
- Can be passed to functions

```
1 import numpy as np
2
3 def improve(update, close, guess=1):
4     while not close(guess):
5         guess = update(guess)
6     return guess
7
8 def update(guess):
9     return 1/guess + 1
10
11 improve(update, close=lambda g: np.allclose(g*g, g+1))
```

1.6180257510729614

Variable scope

- Functions introduce a new namespace
 - Function arguments and variables inside function
- Arguments passed by reference

```
1 x, y = [1, 2], 1
2
3 def f(x):
4     z = 10
5     x.append(3)
6     print(x, y) # Private x, global y
7
8 f(x)
9 print(x, y) # z is not available here
```

```
[1, 2, 3] 1
[1, 2, 3] 1
```

Variable scope ...

```
1 x, y = [1, 2], 1
2
3 def f(x):
4     y = 10
5     x.append(3)
6     print(x, y) # Private x, local y
7
8 f(x)
9 print(x, y)
```

```
[1, 2, 3] 10
[1, 2, 3] 1
```

Some builtin functions

- Available in `builtins` module
- `abs`, `all`, `any`, `ascii`, `bin`, `breakpoint`
- `callable`, `chr`, `compile`, `delattr`, `dir`, `divmod`
- `eval`, `exec`, `format`, `getattr`, `globals`,
- `hasattr`, `hash`, `hex`, `id`, `input`, `isinstance`, `issubclass`, `iter`
- `len`, `locals`, `max`, `min`, `next`, `oct`, `open`, `ord`, `pow`, `print`
- `repr`, `round`, `setattr`, `sorted`, `sum`, `vars`
- Homework: read the docs for each of these!

Modules and imports

- Modules introduce a namespace
- Module/file name should be a valid variable
- Simple modules are just Python source files
- Python extension modules which are compiled native code

```
import module
import module as M
from module import name1
from module import name1, name2
from module import * # Avoid using this!
```

Writing scripts

- Scripts are Python files
- Can use the magic `__name__` variable to guard
 - `__name__` is '`__main__`' when executed as a script
 - `__name__` is the module name when imported

```
if __name__ == '__main__':  
    main()
```

Summary

- Introduced very basic Python syntax
- Will learn a little more later
- Learn language syntax from the [Python tutorial](#)
- See the documentation on the [standard library](#)