Object-oriented programming (in Python)
x, y = 'a', 'b'

def f1():
    x = 1
    print x, y

def f2(y):
    x = 2
    print x, y

f1()
f2(3)
print type(x), type(y)
print x, y
```python
x, y = 'a', 'b'

def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
```

**scopes**
(determined by program code)
```python
x, y = 'a', 'b'
def f1():
    x = 1
    print x, y
def f2(y):
    x = 2
    print x, y
f1()
f2(3)
print type(x), type(y)
print x, y
```

**scopes**
(determined by program code)

**namespaces**
(a snapshot of program execution)
```python
x, y = 'a', 'b'

def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
```

---

**Scopes**

(determined by program code)

**Namespaces**

(a snapshot of program execution)
```python
x, y = 'a', 'b'

def f1():
    x = 1
    print x, y

def f2(y):
    x = 2
    print x, y

f1()
f2(3)
print type(x), type(y)
print x, y
```

**scopes**
(determined by program code)

**namespaces**
(a snapshot of program execution)

**global**
- `x` → `'a'`
- `y` → `'b'`

**built-in**
- `type` → (and others)
```python
file / module / session

``` file

```python
def f1():
    x = 1
    print x, y

f1()
```

```python
def f2(y):
    x = 2
    print x, y

f2(3)
```

```python
x, y = 'a', 'b'
print type(x), type(y)
print x, y
```

---

**scopes**
(determined by program code)

---

**namespaces**
(a snapshot of program execution)

---

**global**

<table>
<thead>
<tr>
<th>x</th>
<th>'a'</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>'b'</td>
</tr>
<tr>
<td>f1</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
</tbody>
</table>

---

**built-in**

<table>
<thead>
<tr>
<th>type</th>
<th></th>
</tr>
</thead>
</table>

(and others)
```python
def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
```
```python
def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)
f1()
f2(3)
print(type(x), type(y))
print(x, y)
```

**scopes** (determined by program code)

**namespaces** (a snapshot of program execution)
```python
x, y = 'a', 'b'
def f1():
    x = 1
    print(x, y)
def f2(y):
    x = 2
    print(x, y)
f1()
f2(3)
print(type(x), type(y))
print(x, y)
```

**scopes**
(determined by program code)

**namespaces**
(a snapshot of program execution)
```python
x, y = 'a', 'b'
def f1():
    x = 1
    print x, y

def f2(y):
    x = 2
    print x, y
f1()
f2(3)
print type(x), type(y)
print x, y
```
```python
x, y = 'a', 'b'

def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
```

**scopes**
(determined by program code)

**namespaces**
(a snapshot of program execution)
```python
file / module / session

x, y = 'a', 'b'
def f1():
    x = 1
    print x, y
def f2(y):
    x = 2
    print x, y
f1()
f2(3)
print type(x), type(y)
print x, y
```

**scopes**
(determined by program code)

**namespaces**
(a snapshot of program execution)
```python
x, y = 'a', 'b'

def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
```

**scopes** (determined by program code)

**namespaces** (a snapshot of program execution)
```python
x, y = 'a', 'b'

def f1():
    x = 1
    print x, y

def f2(y):
    x = 2
    print x, y

f1()
f2(3)
print type(x), type(y)
print x, y
```

scopes
(determined by program code)
Let's practice!

def fact(n):
    if (n == 0):
        return 1
    return n * fact(n-1)

n = 4
result = fact(n / 2)
It’s *all* namespaces!
```python
usefulValue = 42

def printValue(value):
    print value

printValue(usefulValue)
```

(file / module / session)

(useful.py)

(scopes)
(determined by program code)
```python
usefulValue = 42

def printValue(value):
    print(value)

printValue(usefulValue)```

**scopes**
(determined by program code)

**namespaces**
(a snapshot of program execution)
Python

File / Module / Session

1. `import useful`

Scopes
(determined by program code)

Built-in
(type (and others))

Global

useful

usefulValue → 42
printValue →

Namespaces
(a snapshot of program execution)
import useful as other

other →

usefulValue → 42

printValue →

built-in type

(global names) (and others)

(file / module / session)

(scopes) (determined by program code)

(namespaces) (a snapshot of program execution)
```python
from useful import usefulValue

usefulValue → 42
```

**scopes**
(determined by program code)

**built-in**
(type → (and others))

**global**

**namespaces**
(a snapshot of program execution)
A Python idiom
For writing a main function

```python
def main():
    # code for main program goes here

if __name__ == '__main__':
    main()
```
```python
usefulValue = 42

def printValue(value):
    print(value)

def main():
    printValue(usefulValue)

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

**Scopes** (determined by program code)

**Namespaces** (a snapshot of program execution)

**Built-in**

- `type (and others)`

**Global**

- `__name__ = '__main__'`
import useful

__name__ → '__main__'

useful

  __name__ → 'useful'

  usefulValue → 42

  printValue

  main

(type (and others)

built-in)

(scopes
determined by program code)

(namespaces
a snapshot of program execution)
Why object-oriented programming?
What is an object?

• a little state machine!
• combines state and behavior
• separates interface from implementation
• typically has a concept of identity
• is self-contained (and knows about itself)
Object-oriented programming languages differ in:

- how the programmer specifies an object's interface
- how the programmer specifies an object's implementation
- how objects are created, initialized, queried, and updated
- encapsulation mechanism
  how strictly the language enforces the separation between interface & implementation
interface
what a piece of code can do

implementation
how a piece of code works

type
describe a set of supported operations

class
implement a type’s operations

subtype
add more operations to an existing type

subclass
re-use/modify an existing implementation

object

inheritance
in-theory: only about extending implementation
in practice: extends interface and implementation
Object-oriented programming (in Python)
Object-oriented programming languages differ in:

- how the programmer specifies an object's **interface**
- how the programmer specifies an object's **implementation**
- how objects are **created**, **queried**, and **updated**
- **encapsulation** mechanism
  - how strictly the language *enforces* the separation between interface & implementation
It’s *all* namespaces!
class HMCCourse:
    college = 'HMC'
The body of a `class` defines a scope (and it will have a corresponding namespace when the program runs).

`college` is a `class variable` (it belongs to all instances of the class).
```python
class HMCCourse:
    college = 'HMC'
```

- **Scopes (determined by program code)**
- **Namespaces (a snapshot of program execution)**

- **Built-in types** (and others)
  - `type`
We create an instance of a class by “calling” the class.

An instance also defines a scope (and it will have a corresponding namespace when the program runs).
```python
class HMCCourse:
    college = 'HMC'

cs42 = HMCCourse()
```

**Scopes** (determined by program code)

**Namespaces** (a snapshot of program execution)
```python
class HMCCourse:
    college = 'HMC'

cs42 = HMCCourse()
```

Scopes (determined by program code)

Namespaces (a snapshot of program execution)
When we look up a member of an instance (e.g., the `college` member of instance `cs42`), we first look in the namespace of the instance, then in the namespace of its class.
```python
class HMCCourse:
    college = 'HMC'

cs42 = HMCCourse()
cs42.college
```

**Scopes**
(determined by program code)

**Namespaces**
(a snapshot of program execution)
When we define a *method* in Python, we must declare at least one parameter.

By convention, that first parameter is called `self`.

`self` refers to an instance of the class, and we must use `self` to access any members of that instance.
```python
class HMCCourse:
    college = 'HMC'

def printCollege(self):
    print self.college

cs42 = HMCCourse()
cs42.printCollege()
```

When we call a method, Python automatically binds the instance to the `self` parameter.
```python
class HMCCourse:
    college = 'HMC'

    def printCollege(self):
        print(self.college)

cs42 = HMCCourse()
cs42.printCollege()
```

Scopes (determined by program code)

Namespaces (a snapshot of program execution)
```python
class HMCCourse:
    college = 'HMC'

def printCollege(self):
    print self.college

cs42 = HMCCourse()
cs42.printCollege()
```

**Scopes** (determined by program code)

**Namespaces** (a snapshot of program execution)

- **Global:**
  - `college` (determined by program code)
  - `printCollege`

- **Built-in:**
  - `type`
  - `(and others)`

- **Local:**
  - `self`
  - `cs42.printCollege, called @ line 8`
```python
class HMCCourse:
    college = 'HMC'
    def printCollege(self):
        print self.college

cs42 = HMCCourse()
print cs42.printCollege()
```