Lecture 20: Object Oriented Programming
Encapsulation

Comp 102

Forman Christian University

Recap

Last Time: Data Abstraction

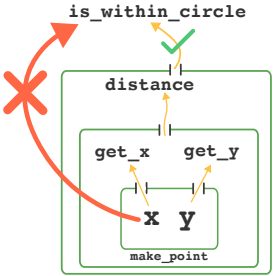
- Separating how data is **represented** from how it is **manipulated**

Last Time: Data Abstraction

- Separating how data is **represented** from how it is **manipulated**
- **Constructors**: create compound data *(e.g.* `rational(n,d)`*)*
- **Accessors**: retrieve components *(e.g.* `numer(x)`*)*

Last Time: Data Abstraction

- Separating how data is **represented** from how it is **manipulated**
- **Constructors**: create compound data *(e.g. `rational(n,d)`)*
- **Accessors**: retrieve components *(e.g. `numer(x)`)*
- **Abstraction Barriers**: higher-level code should **never** bypass constructors/accessors



The Point Abstraction

```
1  def make_point(x, y):
2      return [x, y]           # mutable!
3
4  def get_x(p):
5      return p[0]
6
7  def get_y(p):
8      return p[1]
9
10 def distance(p1, p2):
11     return ((get_x(p1)-get_x(p2))**2 + (get_y(p1)-get_y(p2))**2)**0.5
12
13 def is_within_circle(point, center, radius):
14     return distance(point, center) <= radius
```

The Problem

Everything works through the interface:

```
P = make_point(1, 2)
C = make_point(0, 0)
is_within_circle(P, C, 3)  # True
```

The Problem

Everything works through the interface:

```
P = make_point(1, 2)
C = make_point(0, 0)
is_within_circle(P, C, 3)  # True
```

But **nothing stops** someone from doing this:

```
P[0] = 5  # Abstraction Barrier Violation!
is_within_circle(P, C, 3)  # Wrong answer!
```

Real-World Example

Bank Account:

- You **can't** change the balance directly

Real-World Example

Bank Account:

- You **can't** change the balance directly
- You interact through a well-defined interface:
  - `deposit()`
  - `withdraw()`
  - `get_balance()`

Real-World Example

Bank Account:

- You **can't** change the balance directly
- You interact through a well-defined interface:
  - `deposit()`
  - `withdraw()`
  - `get_balance()`
- **Is there a way to put a lock on the box?**

Big Idea

Encapsulation has two parts:

1. **Information Hiding** — restrict direct access to internal data. Users interact only through a well-defined **interface**.
2. **Bundling** — package data and behavior **together** into a single unit.

Let's tackle **hiding** *first, then* **bundling**.

Higher-Order Functions: Closures

Functions Can Return Functions

```
def make_adder(x):
    def adder(y):
        return x + y
    return adder
```

Functions Can Return Functions

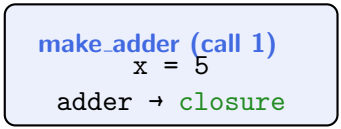
```
def make_adder(x):
    def adder(y):
        return x + y
    return adder

add5 = make_adder(5)
add5(10)          # 15
add5(20)          # 25
```

Tracing make_adder

```
def make_adder(x):
    def adder(y):
        return x + y
    return adder

add5 = make_adder(5)
add5(10)    # 15
```

Tracing make_adder

```
def make_adder(x):
    def adder(y):
        return x + y
    return adder

add5 = make_adder(5)
add5(10)    # 15

add10 = make_adder(10)
add10(10)   # 20
```



add5 remembers x = 5

add10 remembers x = 10

Verify on PythonTutor!

What Is a Closure?

- A **closure** is a function that **remembers** the frame in which it was created

What Is a Closure?

- A **closure** is a function that **remembers** the frame in which it was created
- Even **after** that frame has exited!

What Is a Closure?

- A **closure** is a function that **remembers** the frame in which it was created
- Even **after** that frame has exited!
- Each closure has its **own memory**:
  - `add5` remembers `x = 5`
  - `add10` remembers `x = 10`
  - They are **independent**

You Try!

Given `make_adder` from the previous slide, predict the output:

```
add5 = make_adder(5)
add10 = make_adder(10)

print(add5(3))            # ???
print(add10(3))           # ???
print(add5(add10(1)))     # ???
```

You Try!

Given `make_adder` from the previous slide, predict the output:

```
add5 = make_adder(5)
add10 = make_adder(10)

print(add5(3))           # ???
print(add10(3))          # ???
print(add5(add10(1)))    # ???
```

Solution:

```
print(add5(3))           # 8
print(add10(3))          # 13
print(add5(add10(1)))    # 16   (add10(1)=11, add5(11)=16)
```

Big Idea

Closures **capture** variables from their enclosing scope.

Each call to the outer function creates a **new, independent** closure.

Encapsulation

Let's Lock the Point

Before *(list)*:

```
def make_point(x, y):
    return [x, y]

def get_x(P):
    return P[0]

def get_y(P):
    return P[1]
```

Let's Lock the Point

Before *(list)*:

```
def make_point(x, y):
    return [x, y]

def get_x(P):
    return P[0]

def get_y(P):
    return P[1]
```

After *(closure)*:

```
def make_point(x, y):
    def point():
        return [x, y]
    return point

def get_x(P):
    return P()[0]

def get_y(P):
    return P()[1]
```

Let's Lock the Point

Before *(list)*:

```
def make_point(x, y):
    return [x, y]

def get_x(P):
    return P[0]

def get_y(P):
    return P[1]
```

After *(closure)*:

```
def make_point(x, y):
    def point():
        return [x, y]
    return point

def get_x(P):
    return P()[0]

def get_y(P):
    return P()[1]
```

Key: `point()` creates a **new list** every time it is called. The **original** `x,y` are locked inside the closure.

The Lock Works!

```
P = make_point(1, 2)
get_x(P)                    # 1
```

The Lock Works!

```
P = make_point(1, 2)
get_x(P)                    # 1
```

Try to break in:

```
P()[0] = 6                  # modifies a throwaway list
get_x(P)                    # still 1!
```

The Lock Works!

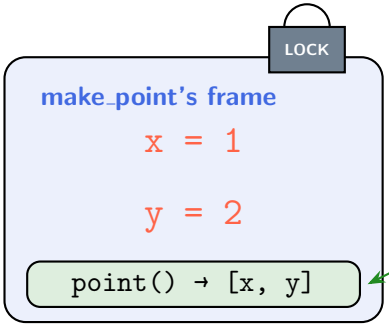
```
P = make_point(1, 2)
get_x(P)                    # 1
```

Try to break in:

```
P()[0] = 6                  # modifies a throwaway list
get_x(P)                    # still 1!
```

The lock works! The closure returns a **new list** each time — the original `x`, `y` are safe inside.

How It Works




```
P = make_point(1,2)

P()
```

returns [1, 2]

(a new list each time)

The Power of Abstraction

These functions **don't change at all**:

```
def distance(p1, p2):
    return ((get_x(p1) - get_x(p2))**2 +
            (get_y(p1) - get_y(p2))**2)**0.5

def is_within_circle(point, center, radius):
    return distance(point, center) <= radius
```

The Power of Abstraction

These functions **don't change at all**:

```
def distance(p1, p2):
    return ((get_x(p1) - get_x(p2))**2 +
            (get_y(p1) - get_y(p2))**2)**0.5

def is_within_circle(point, center, radius):
    return distance(point, center) <= radius
```

Only `make_point`, `get_x`, `get_y` changed.
Abstraction barriers at work!

You Try!

Using the **encapsulated** `make_point`, predict the output:

```
P = make_point(1, 2)
Q = make_point(4, 5)
C = make_point(0, 0)

is_within_circle(P, C, 3)    # ???
P()[0] = 99
get_x(P)                     # ???
```

You Try!

Using the **encapsulated** `make_point`, predict the output:

```
P = make_point(1, 2)
Q = make_point(4, 5)
C = make_point(0, 0)

is_within_circle(P, C, 3)    # ???
P()[0] = 99
get_x(P)                     # ???
```

Solution:

```
is_within_circle(P, C, 3)    # True  (distance ~2.24)
P()[0] = 99                  # modifies a throwaway list!
get_x(P)                     # 1  (x is locked inside)
```

But What If We Need to Change x?

- Game character moves to a new position
- We *want* to update `x` — but through the **interface**

But What If We Need to Change x?

- Game character moves to a new position
- We *want* to update `x` — but through the **interface**
- Python keyword: `nonlocal`
- Allows a nested function to **modify** a variable in its enclosing scope

set_x with nonlocal

```
1 def make_point(x, y):
2     def point():
3         return [x, y]
4     def set_x(new_x):
5         nonlocal x        # modify enclosing scope
6         x = new_x
7     return point, set_x
```

set_x with nonlocal

```
1  def make_point(x, y):
2      def point():
3          return [x, y]
4      def set_x(new_x):
5          nonlocal x        # modify enclosing scope
6          x = new_x
7      return point, set_x

P, set_x = make_point(1, 2)
print(P())          # [1, 2]

set_x(6)            # change x through the interface
print(P())          # [6, 2]
```

Problem: Too Many Functions

What if Point has 10 behaviors?

```
P, set_x, set_y, get_dist, is_in, ... = make_point(1,
    2)
```

Problem: Too Many Functions

What if Point has 10 behaviors?

```
P, set_x, set_y, get_dist, is_in, ... = make_point(1,
    2)
```

- Messy, error-prone
- Hard to remember the order
- Functions are “loose” — not grouped together

Problem: Too Many Functions

What if Point has 10 behaviors?

```
P, set_x, set_y, get_dist, is_in, ... = make_point(1,
    2)
```

- Messy, error-prone
- Hard to remember the order
- Functions are "loose" — not grouped together

There must be a better way...

The Second Half: Bundling

So far we have **information hiding** ✓

But our functions are **scattered**:

- `get_x`, `get_y` — free-standing functions
- `distance`, `is_within_circle` — also separate
- The data (`x`, `y`) lives in the closure, but the **behavior** lives outside

The Second Half: Bundling

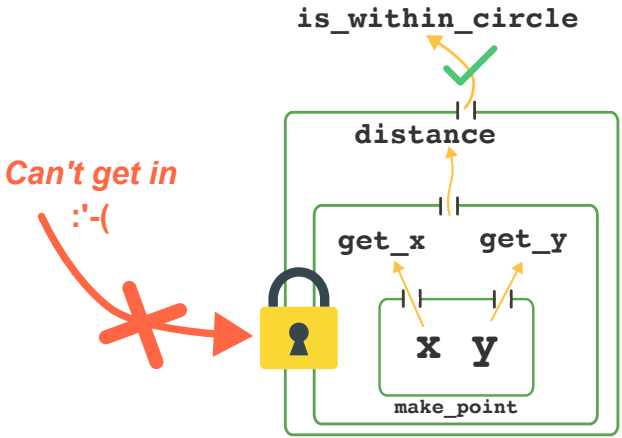
So far we have **information hiding** ✓

But our functions are **scattered**:

- `get_x`, `get_y` — free-standing functions
- `distance`, `is_within_circle` — also separate
- The data (`x`, `y`) lives in the closure, but the **behavior** lives outside

Bundling: package data and behavior **together** into a single unit.

Encapsulation = Hiding + Bundling



Dispatch Dictionary

Attempt 1 (Really Bad)

Put everything inside `make_point`, return only what we need:

```
1  def make_point(x, y):
2      def point():
3          return [x, y]
4      def get_x(P):
5          return P()[0]
6      def get_y(P):
7          return P()[1]
8      def distance(p1, p2):
9          return ((get_x(p1)-get_x(p2))**2 + (get_y(p1)-get_y(p2))**2)**0.5
10     def is_within_circle(pt, center, radius):
11         return distance(pt, center) <= radius
12     return is_within_circle, point
```

Attempt 1 (Really Bad)

Put everything inside `make_point`, return only what we need:

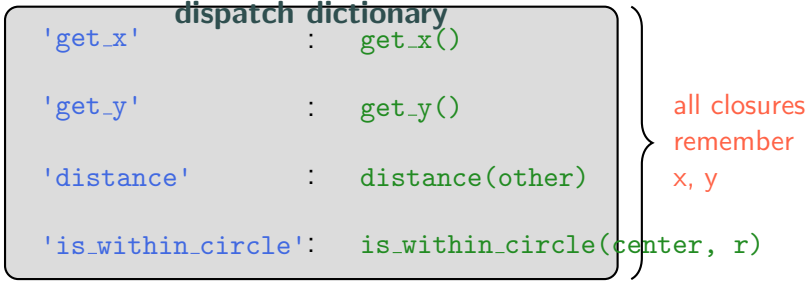
```
1  def make_point(x, y):
2      def point():
3          return [x, y]
4      def get_x(P):
5          return P()[0]
6      def get_y(P):
7          return P()[1]
8      def distance(p1, p2):
9          return ((get_x(p1)-get_x(p2))**2 + (get_y(p1)-get_y(p2))**2)**0.5
10     def is_within_circle(pt, center, radius):
11         return distance(pt, center) <= radius
12     return is_within_circle, point
```

```
is_within_circle, P = make_point(1, 2)
```

OOPS! How do we create another point? Each call creates its **own** copy of all functions!

A Better Way: Dispatch Dictionary

Return a **dictionary** where keys are function names and values are **closures**:



Point with Dispatch Dictionary

```
1  def make_point(x, y):          # Constructor
2      def get_x():                # Closure (Accessor)
3          return x
4      def get_y():                # Closure (Accessor)
5          return y
6      def distance(other):                                  # Closure (Behavior)
7          return ((get_x() - other['get_x']())**2 +
8                  (get_y() - other['get_y']())**2) ** 0.5
9      def is_within_circle(center, radius):                 # Closure (Behavior)
10         return distance(center) <= radius
11
12     dispatch = {
13         'get_x': get_x,
14         'get_y': get_y,
15         'distance': distance,
16         'is_within_circle': is_within_circle
17     }
18     return dispatch
```

Point with Dispatch Dictionary

```
1  def make_point(x, y):          # Constructor
2      def get_x():                # Closure (Accessor)
3          return x
4      def get_y():                # Closure (Accessor)
5          return y
6      def distance(other):                                  # Closure (Behavior)
7          return ((get_x() - other['get_x']())**2 +
8                  (get_y() - other['get_y']())**2) ** 0.5
9      def is_within_circle(center, radius):                 # Closure (Behavior)
10         return distance(center) <= radius
11
12     dispatch = {
13         'get_x': get_x,
14         'get_y': get_y,
15         'distance': distance,
16         'is_within_circle': is_within_circle
17     }
18     return dispatch
```

Note: `get_x()` takes **no argument** — `x` comes from the closure!

Using the Dispatch Dictionary

```
P = make_point(1, 2)
Q = make_point(4, 5)
C = make_point(0, 0)
```

Using the Dispatch Dictionary

```
P = make_point(1, 2)
Q = make_point(4, 5)
C = make_point(0, 0)

P['get_x']()                  # 1
P['is_within_circle'](C, 3) # True
P['distance'](Q)              # 4.24
Q['is_within_circle'](C, 3) # False
```

Using the Dispatch Dictionary

```
P = make_point(1, 2)
Q = make_point(4, 5)
C = make_point(0, 0)

P['get_x']()                  # 1
P['is_within_circle'](C, 3) # True
P['distance'](Q)              # 4.24
Q['is_within_circle'](C, 3) # False
```

Looks like sending a message to an object!

Example: Rational Numbers

```
1  def make_rational(n, d):     # Constructor
2      def get_numer():          # Accessor
3          return n
4      def get_denom():          # Accessor
5          return d
6      def add(other):           # Behavior
7          return make_rational(
8              n * other['get_denom']() + other['get_numer']() * d,
9              d * other['get_denom']())
10     def multiply(other):      # Behavior
11         return make_rational(n * other['get_numer'](), d * other['get_denom']())
12     def print_rat():          # Behavior
13         print(f"{n}/{d}")
14
15     return {'get_numer': get_numer, 'get_denom': get_denom,
16             'add': add, 'multiply': multiply, 'print': print_rat}
```

Example: Rational Numbers

```
1  def make_rational(n, d):     # Constructor
2      def get_numer():          # Accessor
3          return n
4      def get_denom():          # Accessor
5          return d
6      def add(other):           # Behavior
7          return make_rational(
8              n * other['get_denom']() + other['get_numer']() * d,
9              d * other['get_denom']())
10     def multiply(other):      # Behavior
11         return make_rational(n * other['get_numer'](), d * other['get_denom']())
12     def print_rat():          # Behavior
13         print(f"{n}/{d}")
14
15     return {'get_numer': get_numer, 'get_denom': get_denom,
16             'add': add, 'multiply': multiply, 'print': print_rat}
```

```
r1 = make_rational(1, 2)
r2 = make_rational(1, 3)
r3 = r1['add'](r2);     r3['print']()    # 5/6
r4 = r1['multiply'](r2); r4['print']() # 1/6
```

You Try!

Build `make_telecounter()` with dispatch dictionary. **Data**: `count` (starts at 0). **Behaviors**: `increment()`, `reset()`, `get_count()`. **Expose only**: `increment` and `get_count` (not `reset`).

```
tc = make_telecounter()
tc['increment']()
tc['increment']()
tc['increment']()
tc['get_count']()          # 3
tc['reset']()              # KeyError! (not exposed)
```

You Try!

Build `make_telecounter()` with dispatch dictionary. **Data**: `count` (starts at 0). **Behaviors**: `increment()`, `reset()`, `get_count()`. **Expose only**: `increment` and `get_count` (not `reset`).

```
tc = make_telecounter()
tc['increment']()
tc['increment']()
tc['increment']()
tc['get_count']()          # 3
tc['reset']()              # KeyError! (not exposed)
```

Solution:

```
def make_telecounter():
    count = 0
    def get_count():
        return count
    def increment():
        nonlocal count
        count += 1
    def reset():
        nonlocal count
        count = 0
    return {'increment': increment, 'get_count': get_count}
```

The Object Metaphor

Our Dispatch Dicts Are Objects!

They fulfill key properties of objects:

1. **Abstraction** — layers hiding implementation details

Our Dispatch Dicts Are Objects!

They fulfill key properties of objects:

1. **Abstraction** — layers hiding implementation details
2. **Encapsulation** — data locked in closures, accessed only through interface

Our Dispatch Dicts Are Objects!

They fulfill key properties of objects:

1. **Abstraction** — layers hiding implementation details
2. **Encapsulation** — data locked in closures, accessed only through interface
3. **Unique Identity & State** — each `make_point()` call creates independent state

Our Dispatch Dicts Are Objects!

They fulfill key properties of objects:

1. **Abstraction** — layers hiding implementation details
2. **Encapsulation** — data locked in closures, accessed only through interface
3. **Unique Identity & State** — each `make_point()` call creates independent state
4. **Message Passing** — objects communicate via function calls: `P['distance'](Q)` = "P, compute your distance to Q"

Summary

- **Encapsulation**: **bundling** data + behavior, **hiding** internals
- **Closures**: functions that **remember** their enclosing frame
- **Dispatch Dictionary**: a clean **public interface** for objects
- Our objects fulfill key **OOP requirements**:
  - Abstraction, Encapsulation, Identity/State, Message Passing

Big Idea

Data Abstraction says: **separate** representation from use.

Encapsulation says: **enforce** that separation with a lock.

Next lecture: Python's `class` syntax gives us a much cleaner way to build these objects.

Questions?