Objects

Class outline:

- Object-oriented programming
- The class statement
- Class methods
- Instance variables
- Class variables

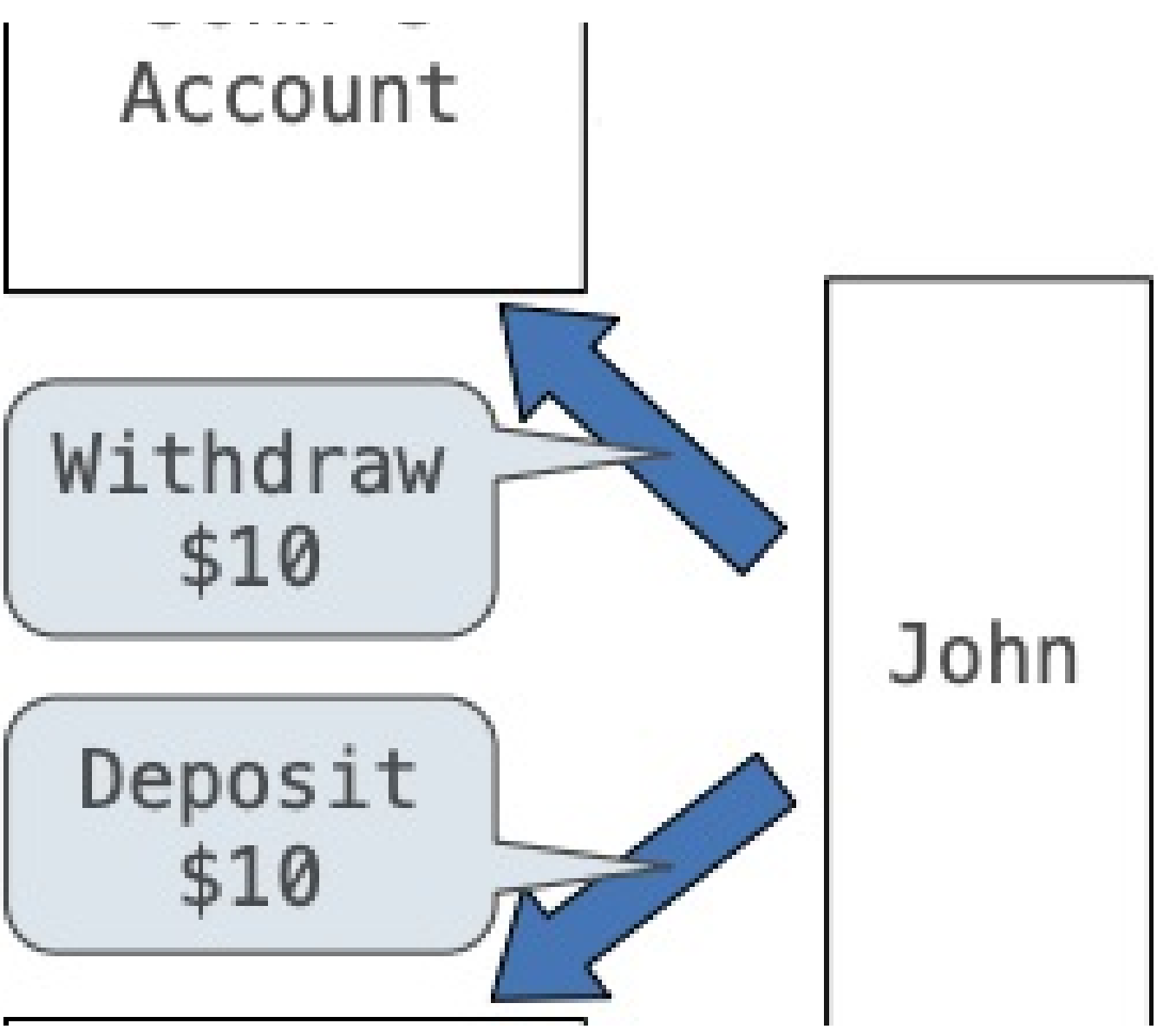
Object-oriented programming

OOP is a method for organizing programs which includes:

- Data abstraction
- Bundling together information and related behavior

A metaphor for computation using distributed state:

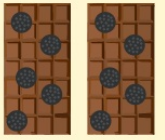
- Each object has its own local state
- Each object also knows how to manage its own local state, based on method calls
- Method calls are messages passed between objects
- Several objects may all be instances of a common type
- Different types may relate to each other


Account
Withdraw
$10
Deposit
$10
John

An OOP shop

Building a chocolate shop

Name: Trufflapagus
Price: $9.99
Nutrition: 170 cals, 19 g sugar
Inventory: 2 bars



Name: Piña Chocolotta
Price: $7.99
Nutrition: 200 cals, 24 g sugar
Inventory: 3 bars

Order #1
Visa

Order #2
Discover

Order #3
AmEx

Name: Coco Lover
Address: 123 Pining St
Nibbsville, OH

Name: Nomandy Noms
Address: 34 Slurpalot Pl
Buttertown, IN

Name: Ammar Chako
Address: 42 Milky Way
Temperville, NV

The OOP approach

We can use objects to organize our code for the shop:

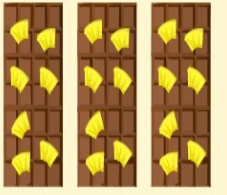
```
# Inventory tracking
Product(name, price, nutrition)
Product.get_label()
Product.get_nutrition_info()
Product.increase_inventory(amount)
Product.reduce_inventory(amount)
Product.get_inventory_report()

# Customer tracking
Customer(name, address)
Customer.get_greeting()
Customer.get_formatted_address()
Customer.buy(product, quantity, cc_info)

# Purchase tracking
Order(customer, product, quantity, cc_info)
Order.ship()
Order.refund(reason)
```

Name: Trufflapagus
Price: $9.99
Nutrition: 170 cals, 19 g sugar
Inventory: 2 bars

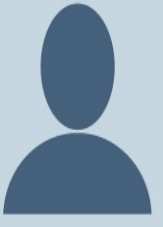
Name: Piña Chocolotta
Price: $7.99
Nutrition: 200 cals, 24 g sugar
Inventory: 3 bars



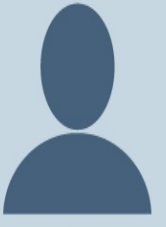
Order #1
Visa

Order #2
Discover

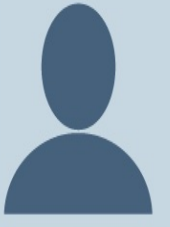
Order #3
AmEx



Name: Coco Lover
Address: 123 Pining St
Nibbsville, OH



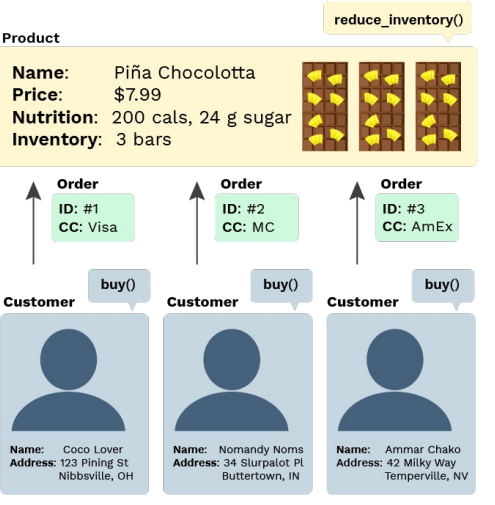
Name: Nomandy Noms
Address: 34 Slurpalot Pl
Buttertown, IN



Name: Ammar Chako
Address: 42 Milky Way
Temperville, NV

Python OOP terminology

- A **class** is a template for defining new data types.
- An instance of a class is called an **object**.
- Each object has data attributes called **instance variables** that describe its state.
- Each object also has function attributes called **methods**.



Python includes special syntax to create classes and objects.

Classes

A fully coded class and usage

```
# Define a new type of data
class Product:

    # Set the initial values
    def __init__(self, name, price, nutrition_info):
        self.name = name
        self.price = price
        self.nutrition_info = nutrition_info
        self.inventory = 0

    # Define methods
    def increase_inventory(self, amount):
        self.inventory += amount

    def reduce_inventory(self, amount):
        self.inventory -= amount

    def get_label(self):
        return "Foxolate Shop: " + self.name

    def get_inventory_report(self):
        if self.inventory == 0:
            return "There are no bars!"
        return f"There are {self.inventory} bars."
```

```
pina_bar = Product("Piña Chocolotta", 7.99,
    ["200 calories", "24 g sugar"])

pina_bar.increase_inventory(2)
```

Let's break it down...

Class instantiation (Object construction)

```
pina_bar = Product("Piña Chocolotta", 7.99,
    ["200 calories", "24 g sugar"])
```

`Product(args)` is often called the **constructor**.

Class instantiation (Object construction)

```
pina_bar = Product("Piña Chocolotta", 7.99,
    ["200 calories", "24 g sugar"])
```

`Product(args)` is often called the **constructor**.

When the constructor is called:

- A new instance of that class is created
- The `__init__` method of the class is called with the new object as its first argument (named `self`), along with any additional arguments provided in the call expression

```
class Product:

    def __init__(self, name, price, nutrition_info):
        self.name = name
        self.price = price
        self.nutrition_info = nutrition_info
        self.inventory = 0
```

Instance variables

Instance variables are data attributes that describe the state of an object.

This `__init__` initializes 4 instance variables:

```
class Product:

    def __init__(self, name, price, nutrition_info):
        self.name = name
        self.price = price
        self.nutrition_info = nutrition_info
        self.inventory = 0
```

The object's methods can then change the values of those variables or assign new variables.

Method invocation

This expression...

```
pina_bar.increase_inventory(2)
```

...calls this function in the class definition:

```
class Product:
    def increase_inventory(self, amount):
        self.inventory += amount
```

Method invocation

This expression...

```
pina_bar.increase_inventory(2)
```

...calls this function in the class definition:

```
class Product:
    def increase_inventory(self, amount):
        self.inventory += amount
```

`pina_bar.increase_inventory` is a **bound method**: a function which has its first parameter pre-bound to a particular value.

In this case, `self` is pre-bound to `pina_bar` and `amount` is set to 2.

Method invocation

This expression...

```
pina_bar.increase_inventory(2)
```

...calls this function in the class definition:

```
class Product:
    def increase_inventory(self, amount):
        self.inventory += amount
```

`pina_bar.increase_inventory` is a **bound method**: a function which has its first parameter pre-bound to a particular value.

In this case, `self` is pre-bound to `pina_bar` and `amount` is set to 2.

It's equivalent to:

```
Product.increase_inventory(pina_bar, 2)
```

Dot notation

All object attributes (which includes variables and methods) can be accessed with **dot notation**:

```
pina_bar.increase_inventory(2)
```

That evaluates to the value of the attribute looked up by `increase_inventory` in the object referenced by `pina_bar`.

The left-hand side of the dot notation can also be any expression that evaluates to an object reference:

```
bars = [pina_bar, truffle_bar]
bars[0].increase_inventory(2)
```

All together now

The class definition:

```
# Define a new type of data
class Product:

    # Set the initial values
    def __init__(self, name, price, nutrition_info):
        self.name = name
        self.price = price
        self.nutrition_info = nutrition_info
        self.inventory = 0

    # Define methods
    def increase_inventory(self, amount):
        self.inventory += amount

    def reduce_inventory(self, amount):
        self.inventory -= amount
```

Object instantiation and method invocation:

```
pina_bar = Product("Piña Chocolotta", 7.99,
    ["200 calories", "24 g sugar"])

pina_bar.increase_inventory(2)
```

Exercise: Player class

```
"""
This class represents a player in a video game.
It tracks their name and health.
"""
class Player:
    """
    >>> player = Player("Mario")
    >>> player.name
    'Mario'
    >>> player.health
    100
    >>> player.damage(10)
    >>> player.health
    90
    >>> player.boost(5)
    >>> player.health
    95
    """
```

Exercise: Player class (solution)

```
"""
This class represents a player in a video game.
It tracks their name and health.
"""
class Player:
    """
    >>> player = Player("Mario")
    >>> player.name
    'Mario'
    >>> player.health
    100
    >>> player.damage(10)
    >>> player.health
    90
    >>> player.boost(5)
    >>> player.health
    95
    """
    def __init__(self, name):
        self.name = name
        self.health = 100

    def damage(self, amount):
        self.health -= amount

    def boost(self, amount):
        self.health += amount
```

Exercise: Clothing class

```
"""
Clothing is a class that represents pieces of clothing in a closet. It tracks the color, category, and cle
"""
class Clothing:
    """
    >>> blue_shirt = Clothing("shirt", "blue")
    >>> blue_shirt.category
    'shirt'
    >>> blue_shirt.color
    'blue'
    >>> blue_shirt.is_clean
    True
    >>> blue_shirt.wear()
    >>> blue_shirt.is_clean
    False
    >>> blue_shirt.clean()
    >>> blue_shirt.is_clean
    True
    """
```

Exercise: Clothing class (solution)

```
"""
Clothing is a class that represents pieces of clothing in a closet. It tracks the color, category, and cle
"""
class Clothing:
    """
    >>> blue_shirt = Clothing("shirt", "blue")
    >>> blue_shirt.category
    'shirt'
    >>> blue_shirt.color
    'blue'
    >>> blue_shirt.is_clean
    True
    >>> blue_shirt.wear()
    >>> blue_shirt.is_clean
    False
    >>> blue_shirt.clean()
    >>> blue_shirt.is_clean
    True
    """

    def __init__(self, category, color):
        self.category = category
        self.color = color
        self.is_clean = True

    def wear(self):
        self.is_clean = False
```

Dynamic attributes

Classes in environment diagrams

```
class Product:

    def __init__(self, name, price, nutrition_info):
    def increase_inventory(self, amount):
    def reduce_inventory(self, amount):
    def get_label(self):
    def get_inventory_report(self):
```

- A class statement creates a new class and binds that class to the class name in the first frame of the current environment.
- Inner `def` statements create attributes of the class (not names in frames).

Visualize in PythonTutor

Dynamic instance variables

An object can create a new instance variable whenever it'd like.

```
class Product:

    def reduce_inventory(self, amount):
        if (self.inventory - amount) <= 0:
            self.needs_restocking = True
        self.inventory -= amount

pina_bar = Product("Piña Chocolotta", 7.99,
    ["200 calories", "24 g sugar"])
pina_bar.reduce_inventory(1)
```

Now `pina_bar` has an updated binding for `inventory` and a new binding for `needs_restocking` (which was not in `__init__`).

 Visualize in PythonTutor

Class variables

Class variables

A **class variable** is an assignment inside the class that isn't inside a method body.

```
class Product:
    sales_tax = 0.07
```

Class variables are "shared" across all instances of a class because they are attributes of the class, not the instance.

```
class Product:
    sales_tax = 0.07

    def get_total_price(self, quantity):
        return (self.price * (1 + self.sales_tax)) * quantity

pina_bar = Product("Piña Chocolotta", 7.99,
    ["200 calories", "24 g sugar"])
truffle_bar = Product("Truffalapagus", 9.99,
    ["170 calories", "19 g sugar"])

pina_bar.sales_tax
truffle_bar.sales_tax
pina_bar.get_total_price(4)
truffle_bar.get_total_price(4)
```

Exercise: StudentGrade class

```
"""
This class represents grades for students in a class.
"""
class StudentGrade:
    """
    >>> grade1 = StudentGrade("Arfur Artery", 300)
    >>> grade1.is_failing()
    False
    >>> grade2 = StudentGrade("MoMo OhNo", 158)
    >>> grade2.is_failing()
    True
    >>> grade1.failing_grade
    159
    >>> grade2.failing_grade
    159
    >>> StudentGrade.failing_grade
    159
    >>>
    """
    def __init__(self, student_name, num_points):
        self.student_name = student_name
        self.num_points = num_points

    def is_failing(self):
        return self.num_points < ___
```

Exercise: StudentGrade class (solution)

```
"""
This class represents grades for students in a class.
"""
class StudentGrade:
    """
    >>> grade1 = StudentGrade("Arfur Artery", 300)
    >>> grade1.is_failing()
    False
    >>> grade2 = StudentGrade("MoMo OhNo", 158)
    >>> grade2.is_failing()
    True
    >>> grade1.failing_grade
    159
    >>> grade2.failing_grade
    159
    >>> StudentGrade.failing_grade
    159
    >>>
    """
    failing_grade = 159

    def __init__(self, student_name, num_points):
        self.student_name = student_name
        self.num_points = num_points

    def is_failing(self):
        return self.num_points < self.failing_grade
```

Accessing attributes

getattr/hasattr built-ins

Using `getattr`, we can look up an attribute using a string

```
getattr(pina_bar, 'inventory')   # 1

hasattr(pina_bar, 'reduce_inventory')  # True
```

`getattr` and dot expressions look up a name in the same way

Looking up an attribute name in an object may return:

- One of its instance attributes, or
- One of the attributes of its class

Public vs. Private

Attributes are all public

As long as you have a reference to an object, you can access or change any attributes.

```
pina_bar = Product("Piña Chocolotta", 7.99,
    ["200 calories", "24 g sugar"])

current = pina_bar.inventory
pina_bar.inventory = 5000000
pina_bar.inventory = -5000
```

You can even assign new instance variables:

```
pina_bar.brand_new_attribute_haha = "instanception"
```

"Private" attributes

To communicate the desired access level of attributes, Python programmers generally use this convention:

- `__` (double underscore) before very private attribute names
- `_` (single underscore) before semi-private attribute names
- no underscore before public attribute names

That allows classes to hide implementation details and add additional error checking.

Quiz: Objects + Classes

Multiple instances

There can be multiple instances of each class.

```
pina_bar = Product("Piña Chocolotta", 7.99,
    ["200 calories", "24 g sugar"])

cust1 = Customer("Coco Lover",
    ["123 Pining St", "Nibbsville", "OH"])

cust2 = Customer("Nomandy Noms",
    ["34 Shlurpalot St", "Buttertown", "IN"])
```

What are the classes here?
How many instances of each?

Multiple instances

There can be multiple instances of each class.

```
pina_bar = Product("Piña Chocolotta", 7.99,
    ["200 calories", "24 g sugar"])

cust1 = Customer("Coco Lover",
    ["123 Pining St", "Nibbsville", "OH"])

cust2 = Customer("Nomandy Noms",
    ["34 Shlurpalot St", "Buttertown", "IN"])
```

What are the classes here? `Product`, `Customer`
How many instances of each? 1 `Product`, 2 `Customer`

State management

An object can use instance variables to describe its state. A best practice is to hide the representation of the state and manage it entirely via method calls.

```
>>> pina_bar = Product("Piña Chocolotta", 7.99,
        ["200 calories", "24 g sugar"])

>>> pina_bar.get_inventory_report()
"There are NO bars!"

>>> pina_bar.increase_inventory(3)
>>> pina_bar.get_inventory_report()
"There are 3 bars total (worth $23.97 total)."
```

What's the initial state?
What changes the state?

State management

An object can use instance variables to describe its state. A best practice is to hide the representation of the state and manage it entirely via method calls.

```
>>> pina_bar = Product("Piña Chocolotta", 7.99,
        ["200 calories", "24 g sugar"])

>>> pina_bar.get_inventory_report()
"There are NO bars!"

>>> pina_bar.increase_inventory(3)
>>> pina_bar.get_inventory_report()
"There are 3 bars total (worth $23.97 total)."
```

What's the initial state? 0 bars in inventory
What changes the state? `increase_inventory()` by changing the instance variable `_inventory`

Class vs. instance variables

```
class Customer:

    salutation = "Dear"

    def __init__(self, name, address):
        self.name = name
        self.address = address

    def get_greeting(self):
        return f"{self.salutation} {self.name},"

    def get_formatted_address(self):
        return "\n".join(self.address)

cust1 = Customer("Coco Lover",
    ["123 Pining St", "Nibbsville", "OH"])
```

What are the class variables?
What are the instance variables?

Class vs. instance variables

```
class Customer:

    salutation = "Dear"

    def __init__(self, name, address):
        self.name = name
        self.address = address

    def get_greeting(self):
        return f"{self.salutation} {self.name},"

    def get_formatted_address(self):
        return "\n".join(self.address)

cust1 = Customer("Coco Lover",
    ["123 Pining St", "Nibbsville", "OH"])
```

What are the class variables? `salutation`
What are the instance variables? `name`, `address`

Python Project of The Day!

Replicate.ai

Replicate.ai: An effort to make machine learning models easy to replicate by anyone..

An example demo using generators (and its source code)