Object-Oriented Programming

Review and Design

Object-Oriented Programming

- **Encapsulation**:
  Classes bundle together related data and functions.
- **Composition**:
  Objects may contain other objects.
- **Inheritance**:
  Objects may inherit behavior from ancestor classes.
- **Polymorphism**:
  A function can run on objects of different classes.

Encapsulation

Bundling together related data and behavior:

```
class Tree:
    """A tree."""
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)

    def __repr__(self):
        if self.branches:
            branch_str = ', ' + repr(self.branches)
        else:
            branch_str = ''
        return 'Tree({0}{1})'.format(self.label, branch_str)

    def __str__(self):
        return '\n'.join(self.indented())

    def indented(self):
        lines = []
        for b in self.branches:
            for line in b.indented():
                lines.append('  ' + line)
        return [str(self.label)] + lines

    def is_leaf(self):
        return not self.branches
```

```
tree = Tree(1, [Tree(1), Tree(2, [Tree(1, [Tree(1)])])])
```

Composition

Objects may contain other objects.

Definitely true for recursive objects:

```
tree = Tree(1, [Tree(1), Tree(2, [Tree(1, [Tree(1)])])])
print(tree.label)
for subtree in tree.branches:
    print(subtree.label)
```

But also true for other objects:

```
class EmissionsTracker:
    def __init__(self, sources=None):
        self.sources = sources or []

    def add_sources(self, sources):
        self.sources.extend(sources_to_add)

tracker = EmissionsTracker()
pp1 = EmissionSource("Anthracite Coal", 2602, 276, 40)
pp2 = EmissionSource("Lignite Coal", 1389, 156, 23)
tracker.add_sources([pp1, pp2])
```

Inheritance

Objects inherit behavior from ancestor classes.

```
class Assignment:
    def __init__(self, title, deadline):
        self.title = title
        self.deadline = deadline

    def __str__(self):
        return f"{self.title} due {self.deadline}"

class Project(Assignment):
    def __init__(self, title, deadline, checkpoints):
        super().__init__(title, deadline)
        self.checkpoints = checkpoints

    def __str__(self):
        return f"{super().__str__()} with checkpoints on {', '.join(self.checkpoints)}"
```

```
lab13 = Assignment("Lab 13", "Apr 27")
scheme = Project("Scheme", "Apr 20", ["Apr 13", "Apr 16"])
print(lab13)
print(scheme)
```

Polymorphism #1

A function can run on objects of different classes.

Easy way: the function runs on any objects that inherit from a particular base class.

```
class Place:
    def add_insect(self, insect):
        insect.add_to(self)

class Insect:
    def add_to(self, place):
        self.place = place

class Bee(Insect):
    pass

class ThrowerAnt(Insect):
    pass

place = Place()
place.add_insect(Bee())
place.add_insect(ThrowerAnt())
```

Polymorphism #2 (🐤Duck typing)

More flexible: a generic function runs on any object that behaves in a particular way.

e.g. functions that run on any iterable (objects with `__iter__`)

```
def print_list(iterable):
    item_num = 1
    for value in iterable:
        print(f"{item_num}. {value}")
        item_num += 1

print_list(["A", "B", "C"])
print_list([x * 3 for x in range(0, 5)])
```

```
class ShoppingList:
    def __init__(self, store, items):
        self.store = store
        self.items = items

    def __iter__(self):
        for item in self.items:
            yield item

shopping_list = ShoppingList("ZeroGrocery", ["Apples", "Tortillas"])
print_list(shopping_list)
```

Polymorphism #3 (Type coercion)

More work: a function converts arguments to the necessary type.

```python
def int_smash(num1, num2):
    """Smashes together positive numbers NUM1 and NUM2, creating
    a number with digits of NUM1 followed by digits of NUM2.
    Non-integers will be converted to integers.
    >>> int_smash(51, 34)
    5134
    >>> int_smash(51.56, 34.72)
    5134
    """
    int1 = int(num1)
    int2 = int(num2)
    num_digits = count_digits(int2)
    while int1 > 0:
        int2 += ( (int1 % 10) * pow(10, num_digits) )
        num_digits += 1
        int1 = int1 // 10
    return int2
```

```python
def count_digits(num):
    num_digits = 0
    while num > 0:
        num_digits += 1
        num = num // 10
    return num_digits
```

Polymorphism #3 (Type coercion)

Another approach to `int_smash`, with even more type coercion:

```
def int_smash(num1, num2):
    """Smashes together positive numbers NUM1 and NUM2, creating
    a number with digits of NUM1 followed by digits of NUM2.
    Non-integers will be converted to integers.
    >>> int_smash(51, 34)
    5134
    >>> int_smash(51.56, 34.72)
    5134
    >>> int_smash('51', '34')
    5134
    >>> int_smash(0x33, 0x22)
    5134
    >>> int_smash(0b110011, 0b100010)
    5134
    """
    return int(num1) * 10 ** len(str(int(num2))) + int(num2)
```

Polymorphism #4 (Type dispatching)

More complexity: the function inspects the argument type to select the appropriate behavior.

```python
def print_obj(obj):
    if hasattr(obj, "__iter__"):
        for item in obj:
            print(item)
    else:
        print(obj)

print_obj([1, 2, 3])
print_obj(123)
```

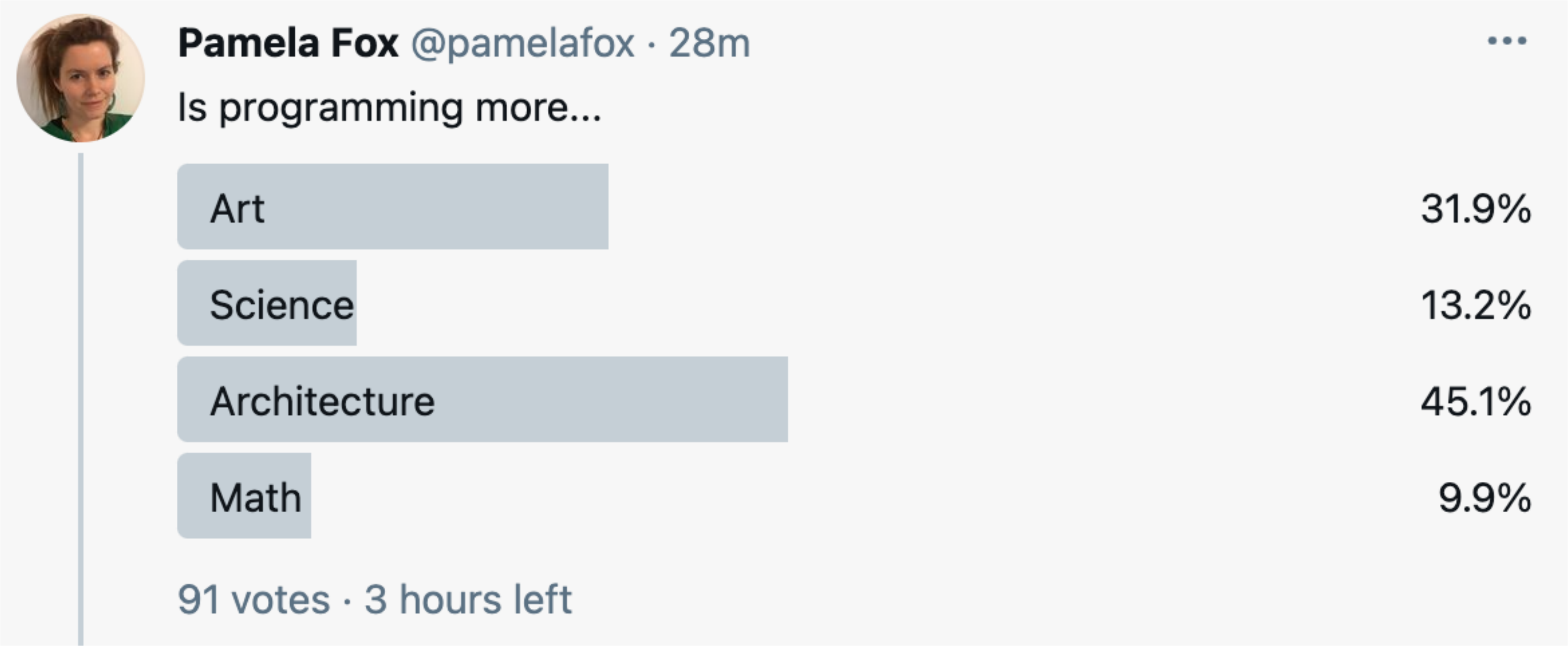
```python
def display_first(data):
    if isinstance(data, Link):
        print(data.first)
    elif isinstance(data, Tree):
        print(data.label)
    else:
        raise Error("Unsupported data type!")

display_first(Link(1, Link(2, Link(3))))
display_first(Tree("A", [Tree("B"), Tree("C")]))
```

Design Principles

⚠️Warning ⚠️

The following slides are not 100% objective!

My own design choices may not be your design choices, or the design choices of your colleagues.



Pamela Fox @pamelafox · 28m

Is programming more...

Art	31.9%
Science	13.2%
Architecture	45.1%
Math	9.9%

91 votes · 3 hours left

Easy Construction

Easy Construction

```
lnk = Link(1, Link(2, Link(3, Link(4))))
```

vs.

```
lnk = LinkedList([1, 2, 3, 4])
```

🤷🏽‍♀️Which do you prefer? 1️⃣ or 2️⃣?

A LinkedList class

```
class LinkedList:

    def __init__(self, values):
        self.head = link = Link(None)
        for value in values:
            link.rest = Link(value)
            link = link.rest

    def __iter__(self):
        link = self.head.rest
        while link is not Link.empty:
            yield link
            link = link.rest
```

```
linked_list = LinkedList([1, 2, 3, 4])
for link in linked_list:
    print(link.first)
```

Set boundaries

Without boundaries 😿

```python
class Insect:
    def __init__(self):
        self.health = 100
        self.perished = False

    def reduce_health(self, amount):
        self.health -= amount
        if self.health <= 0:
            print("Ohno! I have perished.")
            self.perished = True

class Ant(Insect):
    pass

class BumbleBee(Insect):
    def avenge_bee_deaths(self, ant):
        ant.health -= 1000
```

```python
bee = BumbleBee()
ant = Ant()
bee.avenge_bee_deaths(ant)
```

🙊 Oops! Did we just break the game?!

With boundaries 😺

```
class Insect:
    def __init__(self):
        self.__health = 100
        self.__perished = False

    def reduce_health(self, amount):
        self.__health -= amount
        if self.__health <= 0:
            print("Ohno! I have perished.")
            self.__perished = True

class Ant(Insect):
    pass

class BumbleBee(Insect):
    def avenge_bee_deaths(self, ant):
        # ant.__health -= 1000 # 🚫 Error!
        ant.reduce_health(1000)
```

Double underscores prevent accidental access but they can't prevent any access at all, since the attribute is still available at `_classname__attrname`.

Check your assumptions

A Person class

```python
class Person:

    def __init__(self, first_name, middle_name, last_name):
        self.first_name = first_name
        self.middle_name = middle_name
        self.last_name = last_name

    def __str__(self):
        return f"{self.last_name}, {self.first_name} {self.middle_name[0]}."
```

🤔What assumptions does this class make about names? What are examples of names that won't work?

Check assumptions about names

```
p = Person("Ugo", "Tiago Marcondes", "Leal Chaves")
print(p) # Leal Chaves, Ugo T.
p = Person("Vincent", "van", "Gogh")
print(p) # Gogh, Vincent v.
p = Person("Jerome", "K", "Jerome")
print(p) # Jerome, Jerome K.
p = Person("Stephie", "", "Cha")
print(p) # 🚫 Error!
p = Person("Suharto", "", "")
print(p) # 🚫 Error!
p = Person("鄭", "", "根")
print(p) # 🚫 Error!
```

Some names can't be written on a computer at all. Only a fraction of Chinese logograms are represented in the Unicode code points.

Falsehoods Programmers Believe About Names

After: Person

Here's just one way to refactor. However, this refactor needs to be compatible with how the system gets the data about each person, so a UI change may also be needed.

```
class Person:

    def __init__(self, family_name, given_name, family_first=True):
        self.family_name = family_name
        self.given_name = given_name
        self.family_first = family_first

    def __str__(self):
        if self.family_first:
            return f"{self.family_name}, {self.given_name}"
        else:
            return f"{self.given_name} {self.family_name}"
```

- Falsehoods Programmers Believe About Names
- W3C: Personal Names Around the World
- There is No Such Thing as a "Legal Name": A Strange, Shared Delusion

Student/Parent classes

```
class Student(Person):

    def __init__(self, family_name, given_name, mother, father):
        super().__init__(self)
        self.mother = mother
        self.father = father
        self.address = mother.address or father.address

    def __str__(self):
        return f"{super()} (child of {self.mother} and {self.father}"

class Parent(Person):

    def __init__(self, family_name, given_name, address):
        self.address = address
```

🤔What assumptions does this class make about parent/child relations? What are examples of IRL situations that won't work?

Check assumptions about families

- What if they have two mums or two dads?
- What if they have less than 2 parents or more than 2 parents?
- What if the student lives with the other parent?
- What if the student doesn't live with either parent?
- What if the student has an additional guardian that isn't a parent?

Falsehoods Programmers Believe About Families

After: Student/Parent

Here's one refactor that is certainly not perfect.

```
class Student(Person):

    def __init__(self, family_name, given_name, guardians, address):
        super().__init__(self)
        self.guardians = guardians
        self.address = address
        # What could go wrong below?
        self.lives_with_guardian = False
        for guardian in guardians:
            if guardian.address == address:
                self.lives_with_guardian = True

    def __str__(self):
        return f"{super()} (in care of {"".join(self.guardians)}"

class Guardian(Person):

    def __init__(self, family_name, given_name, address):
        self.address = address
```

Address class

```
class Address:

    def __init__(self, street_num, street, apt_or_suite, city, state, zip, country):
        assert street_num > 0
        self.street_num = street_num
        self.street = street
        self.apt_or_suite = apt_or_suite
        self.city = city
        self.zip = zip
        self.country = country

    def __str__(self):
        return f"{self.street_num} {self.street}, {self.apt_or_suite}, \
            {self.city}, {self.state}, {self.country} {self.zip}"
```

```
a = Address(1074, "Live Oaks Blvd", "Apt 1", "Pasadena", "CA", "13078", "US")
print(a)
a = Address(98, "Shirley Street", "", "Pimpama", "QLD", "4209", "Australia")
print(a)
```

🤔What assumptions does this class make about address formats? What are examples of addresses that won't work?

Check assumptions about addresses

```
# No state, city is same as country
a = Address(35, "Mandalay Road", "# 13–37 Mandalay Towers",]
    "Singapore", "", "308215", "Singapore")
print(a)

# No state or postcode
a = Address(150, "Kennedy Road", "Flat 25, 12/F, Acacia Building",
    "Wan Chai", "", "", "Hong Kong Island")
print(a)

# Should actually be written as "101-3485, rue de la Montagne"
a = Address(3485, "rue de la Montagne", "101",
    "Montréal", "Québec", "H3G 2A6", "Canada")
print(a)
```

There are also some addresses we can't construct at all!

Falsehoods Programmers Believe About Addresses

After: Address

Still imperfect, but it's a start.

```
class Address:

def __init__(self, line1, line2, line3, city_or_town,
             state_or_region, zip_or_postcode, country):
    self.line1 = line1
    self.line2 = line2
    self.line3 = line3
    self.country = country
    self.state_or_region = state_or_region
    self.city_or_town = city_or_town
    self.zip_or_postcode = zip_or_postcode

def __str__(self):
    lines = [line for line in [self.line1, self.line2, self.line3] if line]
    newline = '\n'
    return (f"{newline.join(lines)}\n"
        f"{', '.join([self.city_or_town, self.state_or_region])}\n"
        f"{', '.join([self.country, self.zip_or_postcode])}")
```

```
a = Address("101-3485, rue de la Montagne", None, None,
    "Montréal", "Québec", "H3G 2A6", "Canada")
print(a)
```

More ways to check assumptions

All the falsehoods programmers believe in!

General rule: The less your program has to assume about the real world, the better!

Design for Reuse

Before: UCBMFET

```python
class UCBMFET:
    num_members = 0

    def __init__(self, name):
        self.name = name
        self.posts = []
        self.members = []

    def add_member(self, name):
        self.members.append(name)
        UCBMFET.num_members += 1

    def post_in_UCBMFET(self, title_of_post):
        self.posts.append(title_of_post)
```

```python
page = UCBMFET("UCB Memes For Edgy Teens")
page.add_member("Annie")
page.add_member("Grinnell")
page.post_in_UCBMFET("Prepping for 61A Final Be Like...")
```

🤔What would it mean to create another instance of this class? What would that represent? What feels amiss about this design?

After Refactor: MemePage

```python
class MemePage:

    def __init__(self, name, organization):
        self.name = name
        self.organization = organization
        self.posts = []
        self.members = []

    def add_member(self, name):
        self.members.append(name)

    def add_post(self, title_of_post):
        self.posts.append(title_of_post)

    @property
    def num_members(self):
        return len(self.members)
```

```python
page1 = MemePage("UCB Memes For Edgy Teens", "UC Berkeley")
page1.add_member("Annie")
page1.add_member("Grinnell")
page1.add_post("Just Chilling On The Glade")

page2 = MemePage("Wholesome Memes for Tweens", "King Middle School")
page1.add_member("Poppy")
page1.add_member("Sequoia")
page1.add_member("Redwood")
page2.add_post("DEFYING GRAVITY!")
```

Designing reusable classes

The Reuse Test:
🤔Is it possible to create multiple instances of the class, where each instance stores its own relevant state?

👉🏽Use instance variables to store any state that's specific to an instance.

👉🏽Use class variables only for constants or for state that's shared across all instances.