DISCUSSION 07

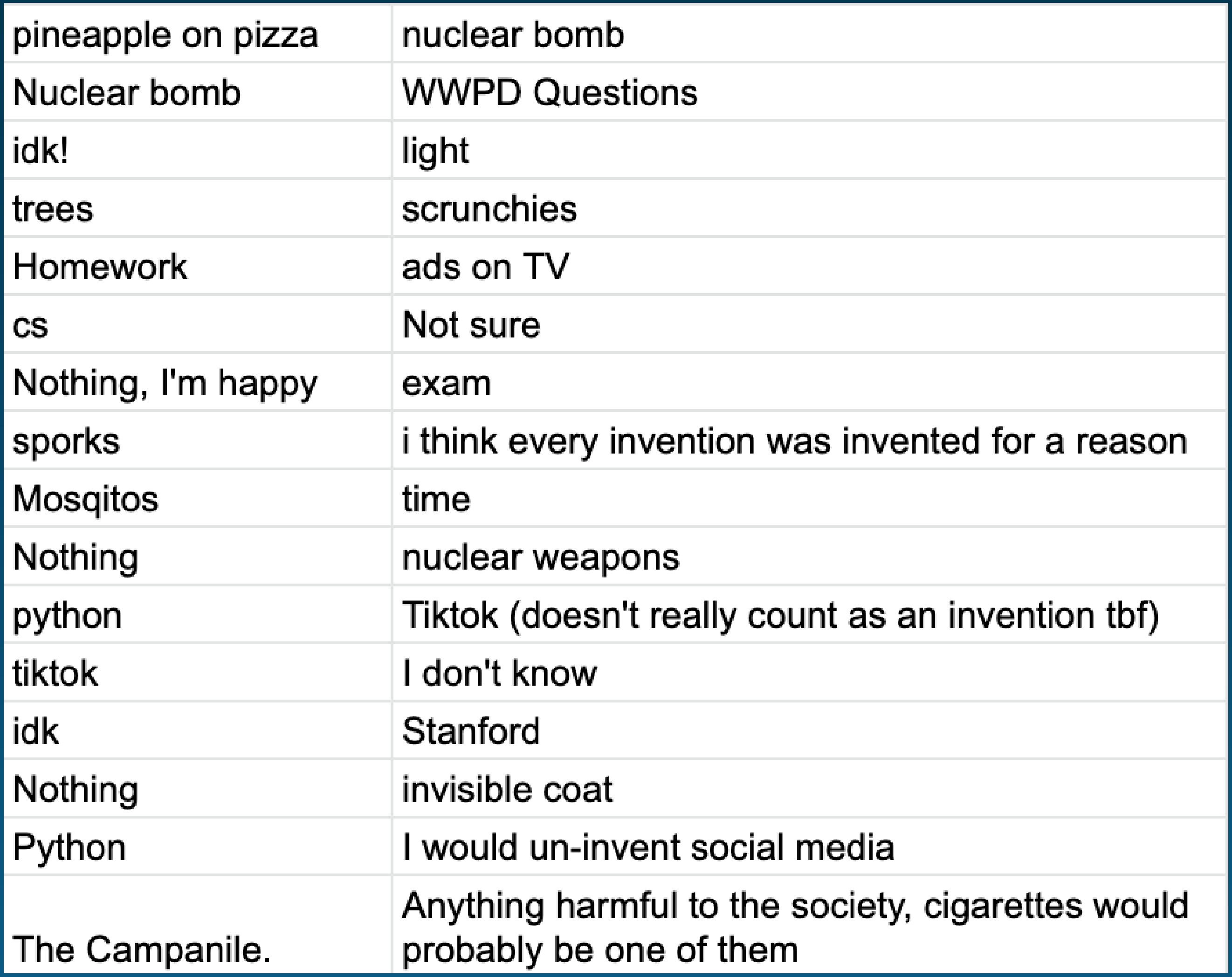
Object-Oriented Programming

Mingxiao Wei
mingxiaowei@berkeley.edu

Mar 9, 2023

FROM LAST TIME... 👀

If you could un-invent one thing, what would it be?



pineapple on pizza	nuclear bomb
Nuclear bomb	WWPD Questions
idk!	light
trees	scrunchies
Homework	ads on TV
cs	Not sure
Nothing, I'm happy	exam
sporks	i think every invention was invented for a reason
Mosqitos	time
Nothing	nuclear weapons
python	Tiktok (doesn't really count as an invention tbf)
tiktok	I don't know
idk	Stanford
Nothing	invisible coat
Python	I would un-invent social media
The Campanile.	Anything harmful to the society, cigarettes would probably be one of them

LOGISTICS 🏡

- Homework 05 due today 03/09
- The ANTS project is coming...
  - If you want to stick with your original project partner / find a new partner, now it's a good time to reach out!
- Reminder - Homework 04 recovery (Ed #1632)
- Exam solution videos by our fav profs 🤩 (Ed #1638)

FROM YOUR MID-SEMESTER FEEDBACK 🧸

- Mini-lecture can be more concise and more engaging
- Go over more questions during lab/discussion
- More time to for you work on problems
- Explain a question using different approaches
- Explain how to apply concepts to problems
- Ends exactly at 11
- More visualization
- More group work

Thanks for all the feedback! I'll definitely try to improve accordingly :)

OBJECT-ORIENTED PROGRAMMING 🥫

OBJECT-ORIENTED PROGRAMMING

OOP - treat code as objects, extending the idea of data abstraction.

- class - a template for objects
- instance - a single object created from a class
- attributes
    - instance variable - specific to an instance
    - class variable - shared by all instances of a class
    - method - a bound function that may be called on all instances of a class
    - Use dot notation to access attributes - `Class.attribute` or `instance.attribute`

OBJECT-ORIENTED PROGRAMMING

```
class Car:
    num_wheels = 4  # class variable, shared by all instances

    def __init__(self, color):  # constructor
        self.wheels = Car.num_wheels
        self.color = color

    def drive(self):  # method
        if self.wheels <= Car.num_wheels:
            return self.color + ' car cannot drive!'
        return self.color + ' car goes vroom!'

    def pop_tire(self):  # method
        if self.wheels > 0:
            self.wheels -= 1
```

TERMINOLOGY

- Attributes = class/instance variables + methods
  - Variables = values (numbers, strings, lists, etc.)
  - Methods = functions defined within a class

	class variable	instance variable
Accessing	`Class.var` or `instance.var` *	`instance.var`
Defining	Within the class, `var = ...`	`instance.var = ...`
Meaning	Shared by all instances of the class	Specific to an instance

```
>>> my_car = Car('red') # an instance of the class
>>> my_car.color # instance variable
'red'
>>> Car.num_wheels, my_car.num_wheels  # both are class variable
4, 4
>>> my_car.wheels # instance variable
4
```

* only works if the instance does not have a instance variable of the same name

MORE TERMINOLOGIES

- Constructors
  - builds an instance of the class
  - define a constructor: `def __init__(self, args):`
  - call a constructor: `ClassName(args)`
  - always returns an *instance* of the class without an explicit return

```
class Car:
    def __init__(self, color):
        self.wheels = Car.num_wheels
        self.color = color
my_car = Car('red')  # create an instance of the Car class
```

- `self.var = ...`
  - Initialize an instance variable `var` for `self` if it doesn't have an instance variable named `var` yet
  - Otherwise update the instance variable `var` for `self` (objects are mutable!)

MORE TERMINOLOGIES

- Methods
  - Functions defined within a class and bound to an instance
  - Think of them as the "verb" of a class
    - a car can *drive* and *pop their tires*

```
>>>  my_car = Car('red')
>>>  my_car.drive()
'red car goes vroom!'
>>>  my_car.wheels
4
>>>  my_car.pop_tire()
>>>  my_car.wheels
3
```

MORE TERMINOLOGIES

- self
  - The first parameter for *nearly* all methods
  - When a method is called, e.g., `instance.method(arg)`, `instance` is *implicitly* bound to `self`, and `arg` corresponds to the rest of the parameters

```
def drive(self):
    if self.wheels <= Car.num_wheels:
        return self.color + ' car cannot drive!'
    return self.color + ' car goes vroom!'
```

```
>>> my_car = Car('red')
>>> my_car.drive()
'red car goes vroom!'
```

CALLING A METHOD

Two equivalent ways of calling a method on an instance:

- `instance.method(...)`
  - `instance` is *implicitly* passed in as the first argument and bound to `self`
- `Class.method(instance, ...)`
  - Need to *explicitly* pass in `instance`

```
>>> my_car = Car('red')
>>> my_car.drive()
'red car goes vroom!'
>>> Car.drive(my_car)
'red car goes vroom!'
```

Either way, a method must be called with dot notation, not on its own!

PROBLEM SOLVING STRATEGIES

- Read the doctests (!!) and the docstring thoroughly to understand the expected behaviors of the class
- Think about what variables are needed to store the information, and how should they be updated
- Pay attention to the expected data type of the attributes (int, string, list, dictionary, another object, etc.)
- Always remember to use dot notation!

WORKSHEET Q1-4

CLASS METHODS 🎒

DECORATORS

- decorator - a function that takes in a function and returns another function (HOF!!)

```python
def f(arg):
    ...
f = decorator(f)
# Above and below are equivalent ways of
# using the decorator
@decorator
def f(arg):
    ...
```

- The `@decorator` syntax is a syntax sugar

CLASS METHODS

```
@classmethod
def method(cls, args):   # defined within a class
    ...
```

- The `@classmethod` decorator turns a method into a class method
- Receive `cls` (the class itself) as the first argument, instead of `self`
- To call a class method, use `Class.method(args)`
  - Use dot notation with the class name
  - No need to specify `cls`
- "factory methods" - construct and return instances of the class

CLASS METHODS

```
class Dog:

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

  @classmethod
    def robo_factory(cls, owner):
        return cls("RoboDog", owner)
```

With `Dog.robo_factory(owner_name)`, we can create a Dog instance with the name `"RoboDog"` whose owner has the name `owner_name`, without having to call the Dog constructor with the dog name `"RoboDog"` every time (`Dog("Robodog", owner_name)`)

WORKSHEET Q5

ATTENDANCE! 🤠

go.cs61a.org/mingxiao-att

- The attendance form and slides are both linked on our section website!
- Please leave any anonymous feedback here go.cs61a.org/mingxiao-anon
- Please do remember to fill out the form by midnight today!!