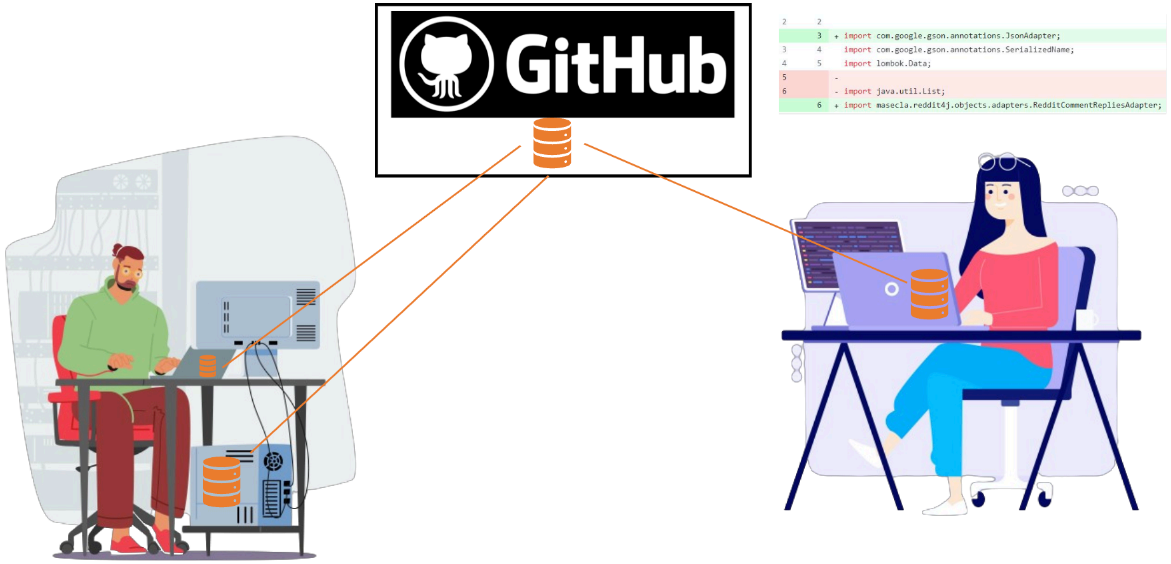
Visibility and Immutabililty

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Git Review



```
+ import com.google.gson.annotations.JsonAdapter;
  import com.google.gson.annotations.SerializedName;
  import lombok.Data;
-
- import java.util.List;
+ import masecla.reddit4j.objects.adapters.RedditCommentRepliesAdapter;
```

When you "start" an assignment using Pawtograder, it creates a code repository on GitHub containing the "starter code" for that assignment.

You then use `git clone` to copy that code repository to your laptop, and open / edit the code using VSCode on your laptop.

Key git concepts

- Repository (repo): a set of code and its history
    - local: on your computer
    - remote: on another computer (like GitHub)
- Commit
    - the codebase at a given point in time (noun)
    - to add a set of changes to the repository (verb)
    - Push: to move code from a local to remote repository

Tip: git repositories have a directory in them called `.git` which is invisible by default. To get it to show up when you use the `ls` command, you must do this: `ls -al`

Locations of versions of code

Location	Definition	git command to put code there	Postal analogy
working area	code that you are currently writing / saving in VSCode		Writing on a paper
staging area	code that is ready to be commited	`git add .`	Add a stamp and put it in your backpack
local repository	code that has been committed	`git commit -m "description"`	Put all stamped cards in the mailbox
remote repository	code on GitHub	`git push`	Workers move cards to destinations

Most assignments will go like this:

- Accept the assignment on Pawtograder
- Pawtograder creates your GitHub repo.
- Using your command line, navigate to this course's directory and do `git clone <GitHub repo URL (ssh version, not html)>`
- You open the resulting files using VSCode and work on the assignment, saving as you go.
- After each significant chunk of progress on the assignment:
  - `git add .` to stage changes in all files in this directory
  - `git commit -m "descriptive message"` to commit the changes
  - `git push` to push the changes to the online repo in GitHub
    - Pawtograder will automatically take that as your submission (if the submission is still open)

Other commands

- `git pull` takes changes any that others pushed to the repo on GitHub, and copies them to your local repo
- `git status` reports which files have been changed and staged
- `git diff` shows every changed line
- `git diff --staged` shows the difference between staged and committed changes
- `history` shows the history of commands you typed into the command line

Poll: Why is this not an ideal commit message? `"Complete Homework 3"`

1. It doesn't describe the code changes
2. It implies that all changes to the entire assignment were submitted in a single commit
3. We can't change Homework 3 again, since we said we completed it
4. All of the above
5. (1) and (2) only

Next new topic:

Visibility

"Unenforced guidelines and best practices" is a recurring theme in this course.

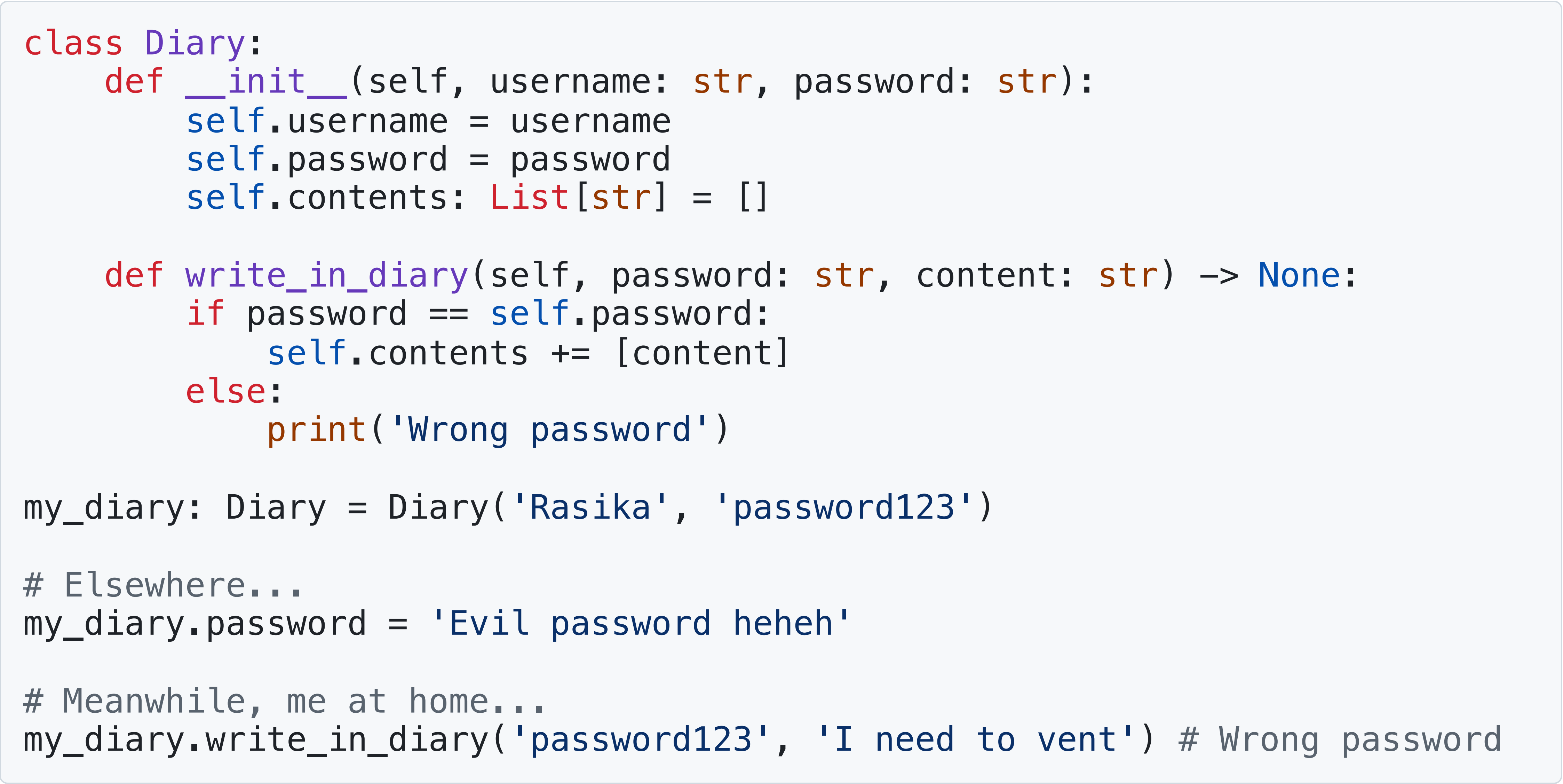
Python's built-in visibility restrictions

Most programming languages have a way to mark attributes and methods as "private" -- they can only be accessed from within their class

- E.g., block access to `my_diary.password`
- E.g., prevent calling the method `car.spray_gas_from_tank_into_cylinders()`

> "Python doesn't do that. Python doesn't really believe in enforcing laws that might someday get in your way. Instead, it provides unenforced guidelines and best practices. Technically, all methods and attributes on a class are publicly available."
> -- Dusty Philips (the author of our suggested textbook)

In Python, we cannot prevent external code from viewing or changing an object's attributes (or using its methods).



```python
class Diary:
    def __init__(self, username: str, password: str):
        self.username = username
        self.password = password
        self.contents: List[str] = []

    def write_in_diary(self, password: str, content: str) -> None:
        if password == self.password:
            self.contents += [content]
        else:
            print('Wrong password')

my_diary: Diary = Diary('Rasika', 'password123')

# Elsewhere...
my_diary.password = 'Evil password heheh'

# Meanwhile, me at home...
my_diary.write_in_diary('password123', 'I need to vent') # Wrong password
```

However: we put an underscore in the name to nicely ask others to avoid using it

```python
class Diary:
    def __init__(self, username: str, password: str):
        self.username = username
        self._password = password
        self.contents: List[str] = []

    def write_in_diary(self, password: str, content: str) -> None:
        if password == self._password:
            self.contents += [content]
        else:
            print('Wrong password')

my_diary: Diary = Diary('Rasika', 'password123')

# Elsewhere...
my_diary._password = "Intentionally accessing what I shouldn't"

# Meanwhile, me at home...
my_diary.write_in_diary('password123', 'I need to vent')
# Still wrong password, but we asked them nicely not to do this
```

Underscore (`_`) in the name: nicely ask others to avoid using it

> "Most Python programmers will interpret this as, 'This is an internal variable, think three times before accessing it directly'. But there is nothing stopping them from accessing it if they think it is in their best interest to do so. Yet, if they think so, why should we stop them? We may not have any idea what future uses our classes may be put to."
> -- Dusty Philips

Understood convention:

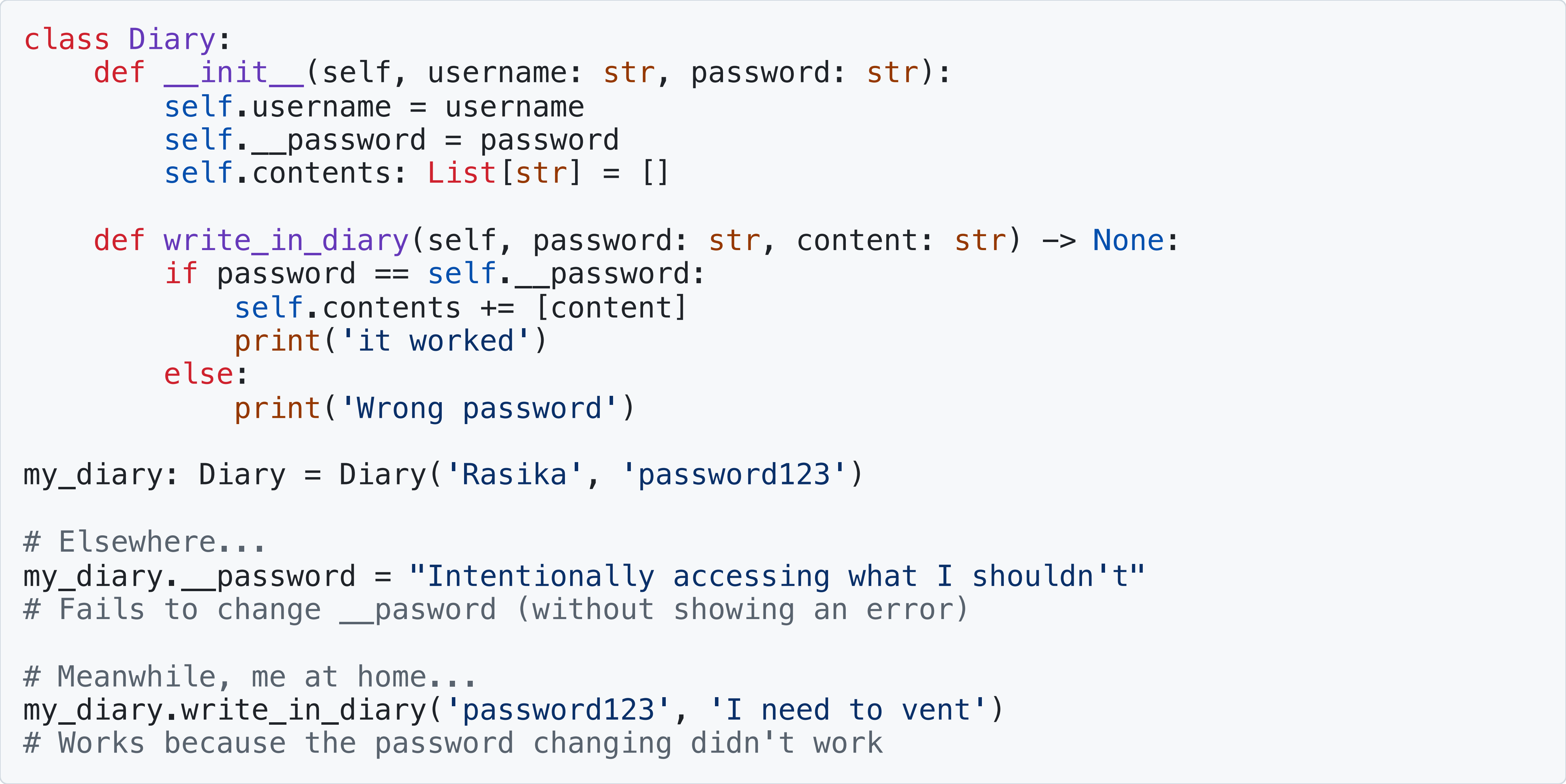
Though we *can* access an attribute named with an underscore, the commonly understood *convention* is that we don't touch it.

It also gets flagged by our linter.

We can ask nicely, or we can strongly suggest it

- Underscore (`_`) in the name: nicely ask others to avoid using it
- *Two* underscores (`__`) in the name: even stronger suggestion to keep away

A double underscore (`__`) *mangles* the name of the attribute: if we want to access it from outside the class, we have to add `_<classname>` to the attribute.



```
class Diary:
    def __init__(self, username: str, password: str):
        self.username = username
        self.__password = password
        self.contents: List[str] = []

    def write_in_diary(self, password: str, content: str) -> None:
        if password == self.__password:
            self.contents += [content]
            print('it worked')
        else:
            print('Wrong password')

my_diary: Diary = Diary('Rasika', 'password123')

# Elsewhere...
my_diary.__password = "Intentionally accessing what I shouldn't"
# Fails to change __pasword (without showing an error)

# Meanwhile, me at home...
my_diary.write_in_diary('password123', 'I need to vent')
# Works because the password changing didn't work
```

Note: it won't raise an error -- it will just silently fail to access the password attribute.

Use the name mangling: add `_Diary` when accessing from outside the class

```python
class Diary:
    def __init__(self, username: str, password: str):
        self.username = username
        self.__password = password
        self.contents: List[str] = []

    def write_in_diary(self, password: str, content: str) -> None:
        if password == self.__password:
            self.contents += [content]
            print('it worked')
        else:
            print('Wrong password')

my_diary: Diary = Diary('Rasika', 'password123')

# Elsewhere...
my_diary._Diary__password = "Intentionally accessing what I shouldn't"
# Works despite MyPy and Pylint complaints

# Meanwhile, me at home...
my_diary.write_in_diary('password123', 'I need to vent')
# Wrong password
```

> "Name mangling does not guarantee privacy, it only strongly recommends it. Most Python programmers will not touch a double-underscore variable on another object unless they have an extremely compelling reason to do so. However, most Python programmers will not touch a single-underscore variable without a compelling reason either."
> -- Dusty Philips

Poll: Which ONE does NOT make it print `EVIL`?

```python
class Grades:
    def __init__(self, student_id: str):
        self._student_id = student_id
        self.__grades: List[int] = [72, 46]

my_grades: Grades = Grades('S999999')
```

1.

```python
my_grades._student_id = 'EVIL'
print(my_grades._student_id)
```

2.

```python
my_grades.__grades.append(-1)
if -1 in my_grades.__grades:
    print('EVIL')
```

3.

```python
my_grades._Grades__grades.append(-1)
if -1 in my_grades._Grades__grades:
    print('EVIL')
```

4.

```python
print(Grades('EVIL')._student_id)
```

If it's not an attribute (it's a property), it can't be modified

Encapsulation via Properties

Allow external code to modify or access "attributes," but only in a controlled way

- E.g., a `Person` has "attribute" `age: int`, which can be modified, but not to a negative number
- E.g., When we access the `Person`'s `name`, it returns the concatenation of the person's `_first_name` and `_last_name`

To create an attribute that is re-calculated every time someone accesses it, we use the `@property` decorator.

The name of the method marked with the `@property` decorator becomes the name of this attribute.

If it's not an attribute (it's a property), it can't be modified

Encapsulation via Properties

Looks like an attribute called `name`, but is actually the concatenation of two attributes:

```python
class Person:
    def __init__(self, first_name: str, last_name: str):
        self._first_name = first_name
        self._last_name = last_name

    @property
    def name(self) -> str:
        return f'{self._first_name} {self._last_name}'

mini: Person = Person('Mini', 'Bhalerao')
print(mini.name) # Mini Bhalerao
```

`_first_name` and `_last_name` can still be accessed, though it is discouraged.

Properties can be modified, but only through a method that we write and control.

Encapsulation via Properties

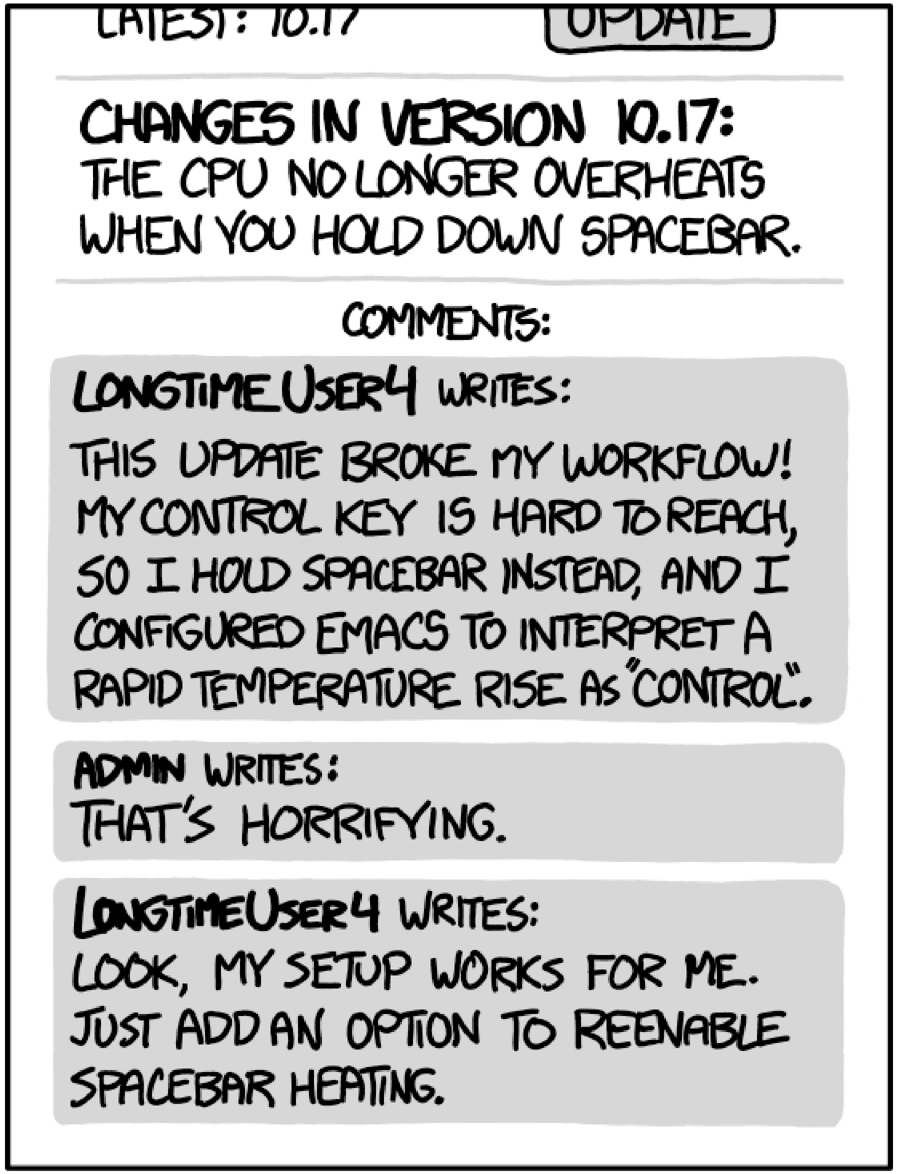
"Attribute" which can only be modified in a controlled way (no negative `age` s):

1. First create the "attribute" using the `@property` decorator
2. Then add a "setter" using another method with the same name, with the decorator `@age.setter`

```python
class Person:
    def __init__(self, age: int):
        self._age = age

    @property
    def age(self) -> int:
        return self._age

    @age.setter
    def age(self, new_age: int) -> None:
        if new_age >= 0:
            self._age = new_age


mini: Person = Person(10)
mini.age = 11
print(mini.age) # 11
```

Poll: Why do we have the decorators `@property` and `@*.setter`?

1. `@property` controls the way outsiders view an attribute
2. `@property` prevents hackers from accessing an attribute
3. `@*.setter` controls the way outsiders can modify an attribute
4. `@*.setter` prevents hackers from modifying an attribute


LATEST: 10.17
UPDATE
CHANGES IN VERSION 10.17:
THE CPU NO LONGER OVERHEATS WHEN YOU HOLD DOWN SPACEBAR.
COMMENTS:
LONGTIME USER4 WRITES:
THIS UPDATE BROKE MY WORKFLOW! MY CONTROL KEY IS HARD TO REACH, SO I HOLD SPACEBAR INSTEAD, AND I CONFIGURED EMACS TO INTERPRET A RAPID TEMPERATURE RISE AS "CONTROL".
ADMIN WRITES:
THAT'S HORRIFYING.
LONGTIME USER4 WRITES:
LOOK, MY SETUP WORKS FOR ME. JUST ADD AN OPTION TO REENABLE SPACEBAR HEATING.

EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

Hyrum's Law:

> "All observable behaviors of your system will be depended on by somebody."

https://www.hyrumslaw.com

https://xkcd.com/1172

Immutability

An object is "immutable" if it cannot be modified after creation.

Lists are not immutable, because they can be modified after creation. Lists are mutable.

```python
my_list: List[int] = [1, 2, 3]
my_list.append(-400)
print(my_list) # [1, 2, 3, -400]
```

Tuples are immutable.

```python
my_tuple: Tuple[int, int, int] = (1, 2, 3)
my_tuple.append(4) # impossible
```

Immutability

We can sort a list in-place, but not a tuple.

```
my_list.sort()
print(my_list) # [-400, 1, 2, 3]

my_tuple.sort() # impossible
```

The tricky part:

- The tuple is immutable
- The variable `my_tuple` (the pointer to the location in the computer's memory) is a mutable variable

We can't mutate the tuple, but we can re-assign the variable `my_tuple` to a sorted version of the same tuple:

```
my_long_tuple: Tuple[int, ...] = tuple([-i for i in range(5)])
my_long_tuple = tuple(sorted(my_long_tuple))
print(my_long_tuple)      # (-4, -3, -2, -1, 0)
```

Poll: Which ONE is not allowed? (Hint: `str`s are immutable)

```
my_str: str = 'mini'
```

1. `print(my_str.upper())`
2. `my_str = my_str.upper()`
3. `my_str = 'MINI'`
4. `my_str[0] = 'B'`

Poll:

1. What is your main takeaway from today?
2. What would you like to revisit next time?