Iterators, Comprehensions and Observers

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 5

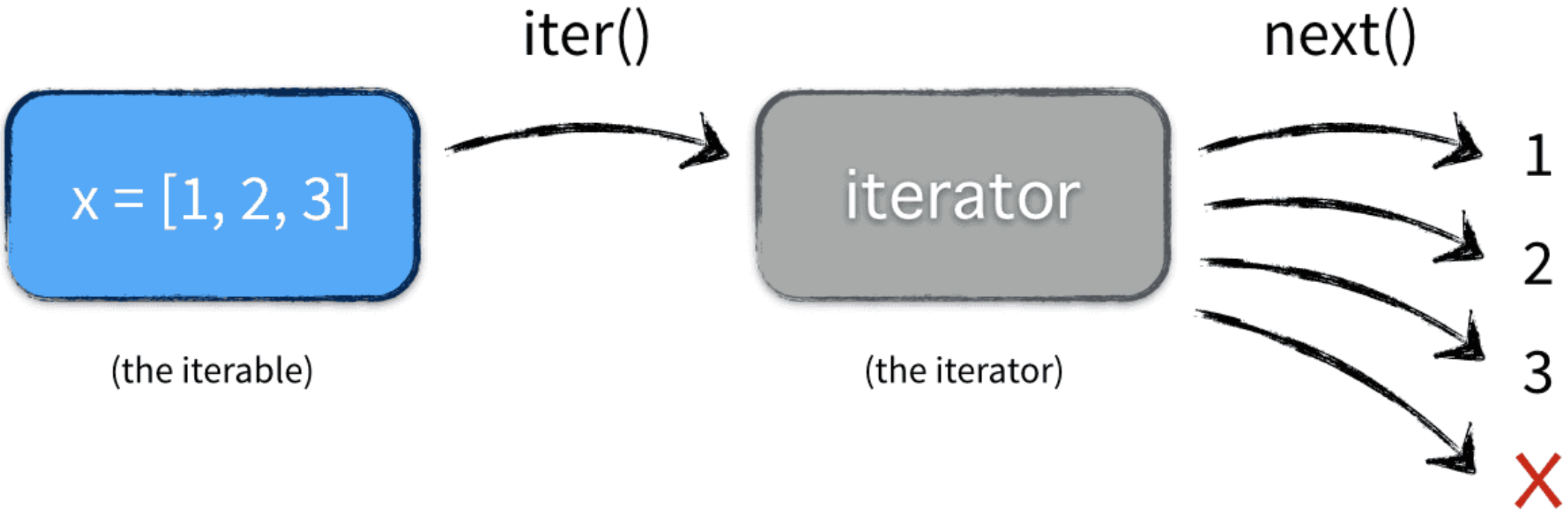
Recap

- Exceptions
- Try-Except-Else-Finally
- Unit Testing
- File handling

Previously On...

COMP 3522

Recap: Iterator and Iterable



Iterators, Iterating and Iterables

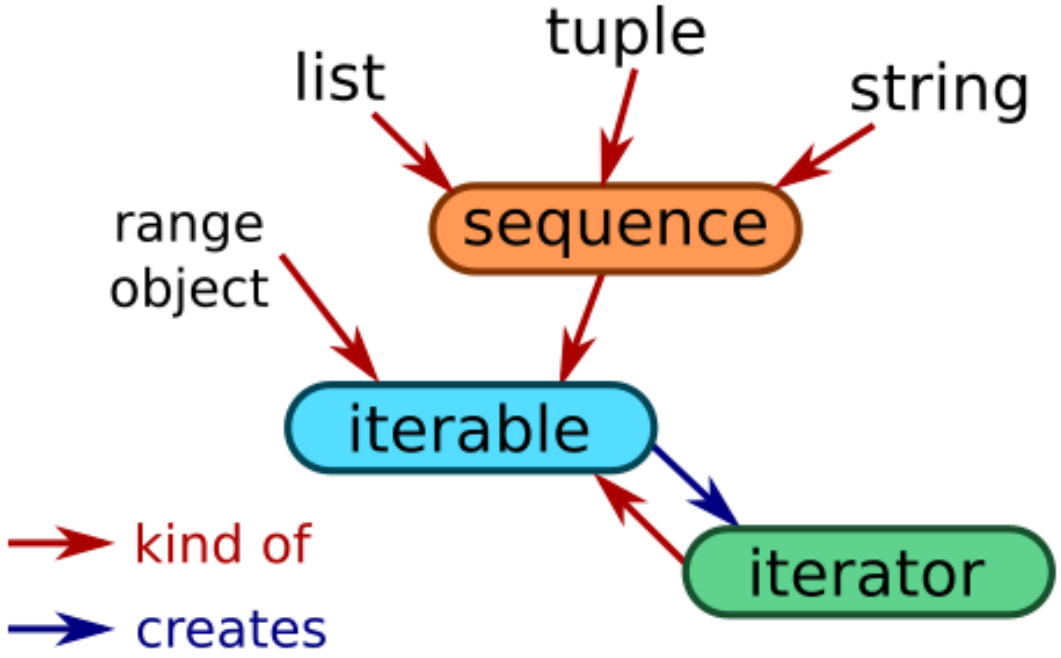
Iterator vs Iterables

An **Iterator** is responsible for traversing an **Iterable**

An **iterators** primary responsibility is to **iterate** over the elements of an **iterable.**

An Iterable is something that you can loop over.

- List
- Dictionary
- Tuples
- Sets
- Any container or sequence really
- Custom data types



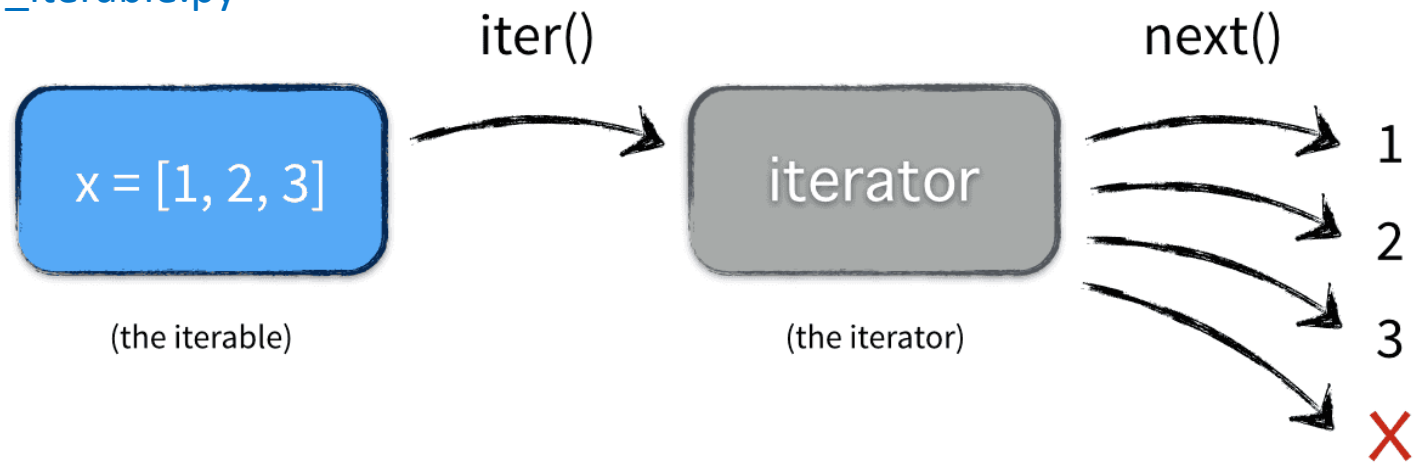
How would we implement this from scratch in Python? (The Iterator Protocol)

Say we wanted to iterate over each word of a sentence and print out the word with its first letter Capitalized.

How would we implement this in python?

We would need to implement an **iterable class** and an **iterator class**.

Let's look at some code: iterator_iterable.py



Iterator and Iterable Protocol

Iterator:

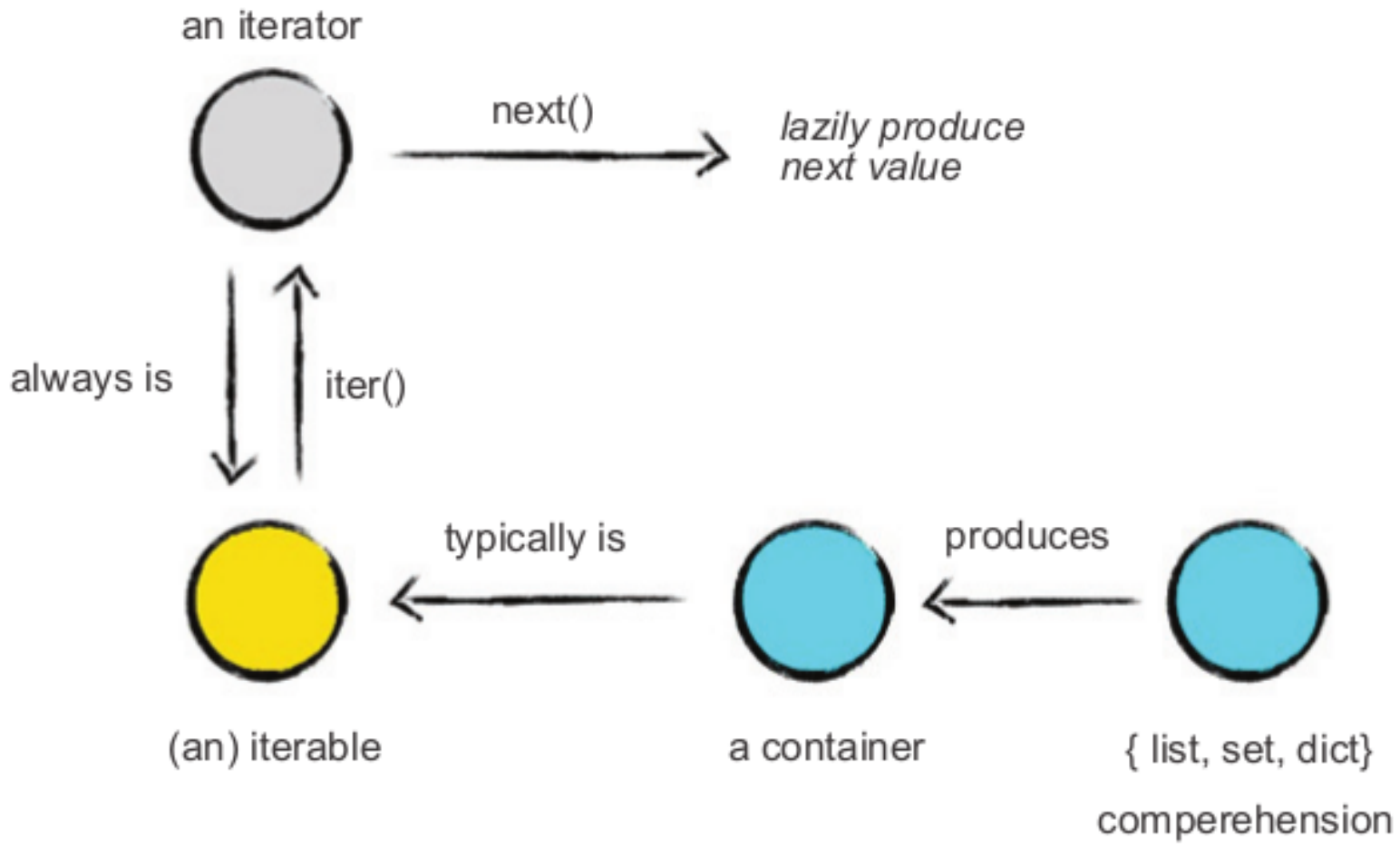
- **Implements __next__(self)**
- Implements __iter__(self)

Iterable:

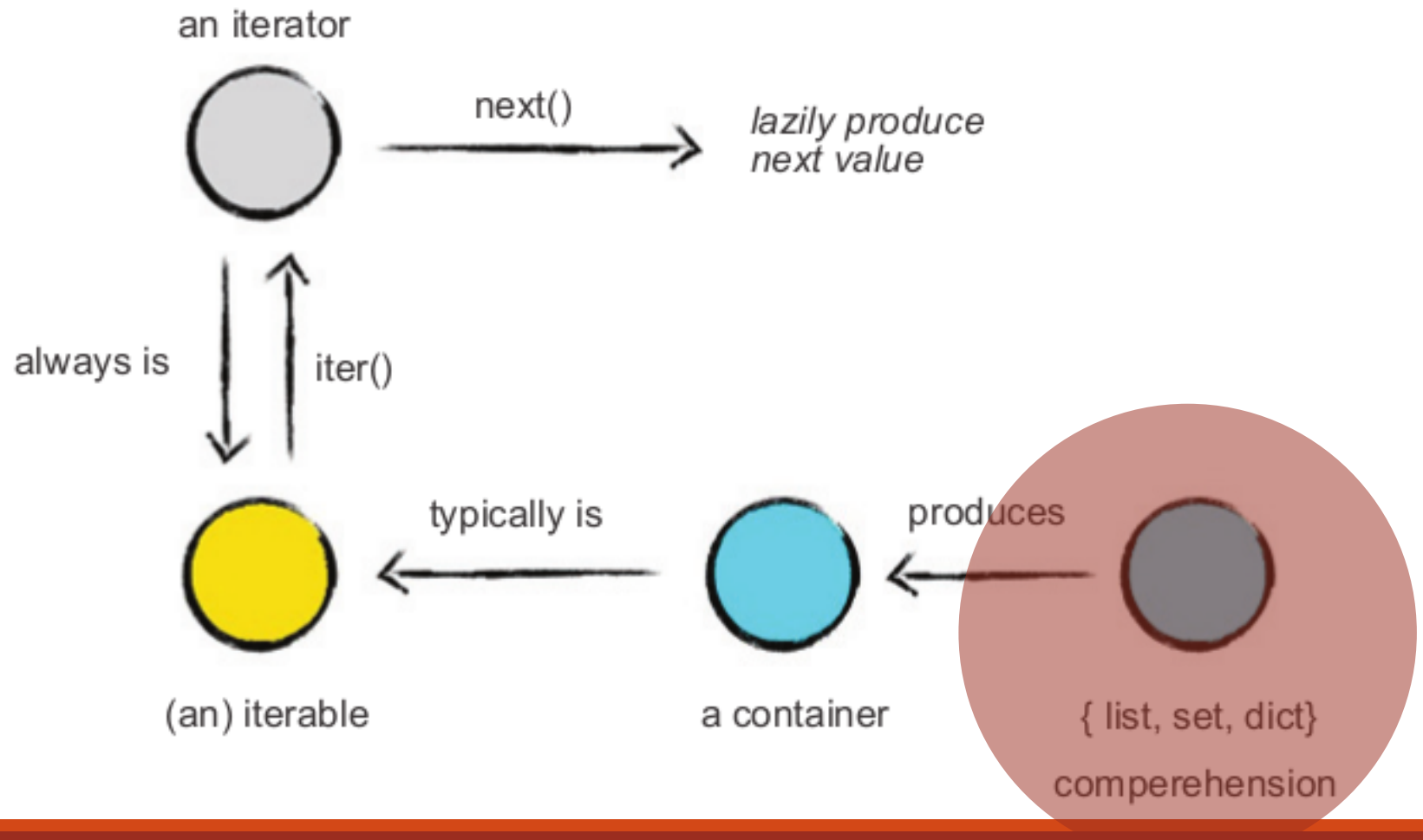
- Implements __iter__(self)

Comprehensions

So this is what we did in the code we just saw



So this is what we did in the code we just saw



What on earth are these?

Comprehensions

Problem: Write code that takes a range of numbers and generates a list that contains all the same numbers, except that the odd numbers are now replaced with ‘x’.

```
list = []
for x in range(0,10):
    if x % 2 == 0:
        list.append(x)
    else:
        list.append('x')
```

That’s a lot of code to do a relatively simple task. **Comprehensions** allow us to compress those 6 lines of code into one line of code

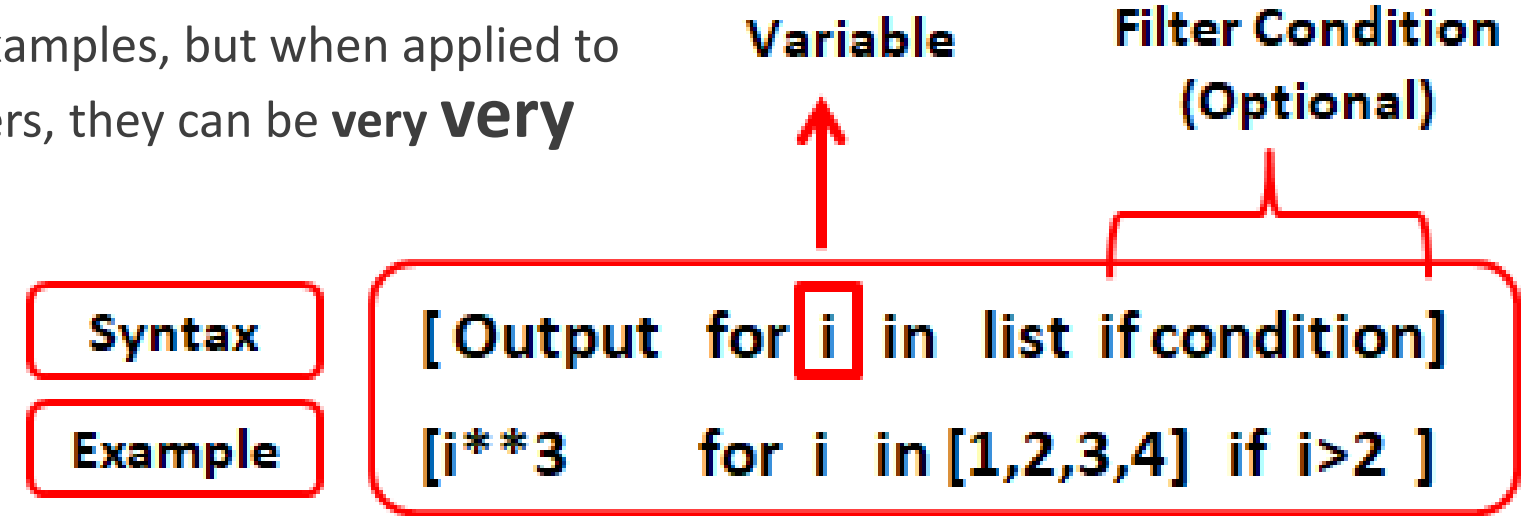
Comprehensions

Comprehensions allow us to transform/filter an iterable object (such as UpperCaseIterable). The resultant object will be another iterable.

In simple words, **create one iterable from another iterable** (sequence/container) by **following some special rules/filters**.

Simple, readable syntax

May seem irrelevant in simple examples, but when applied to complex sequences and containers, they can be **very very powerful**.

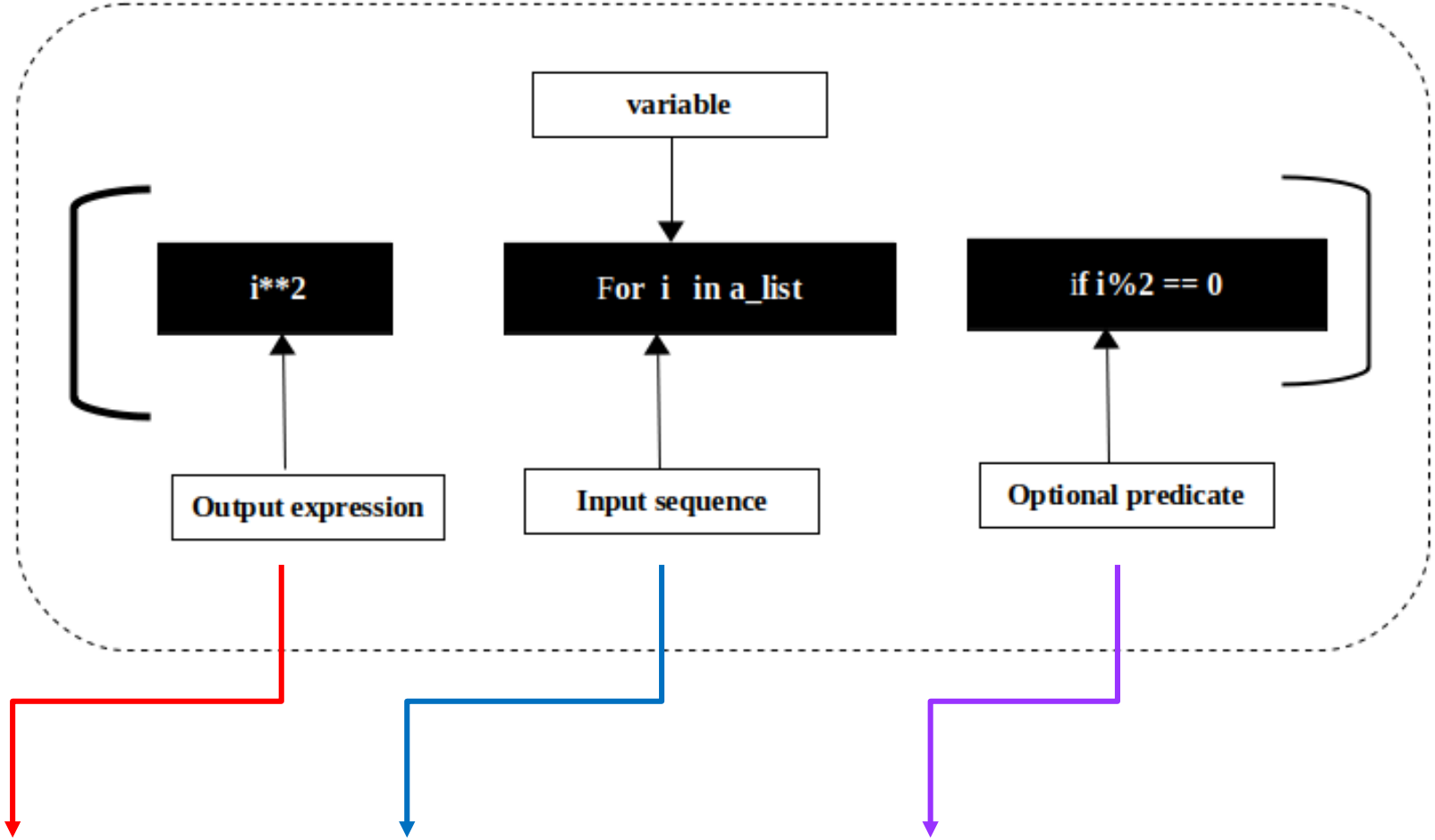


Syntax

Say we have the following iterable:

groceries = [“Bread”, “Milk”, “Mangoes”, “Mandarins”, “Onions”]

We could print a sequence of all the elements starting with ‘M’:



groceries_m = [item for item in groceries if item[0] == ‘M’]

Challenge Time

Challenge 1:

Write a list comprehension that takes a list of strings and generates a list of strings

```
strings = ['hi','hello','bye’]

new_strings = ???
```

Challenge Time

Challenge 2:

Write a list comprehension that takes in numbers in the range of 0-26 (excluding 26) and generates the all the letters in the alphabet (uppercase)

HINT: chr(integer) converts an int to a character

```
alphabet = ???
```

Challenge Time

Challenge 3:

Write a list comprehension that takes a range of numbers and generates a list of all the even numbers in that range

```
even = ???
```

Challenge Time

Challenge 4:

Write a single list comprehension that takes a range of numbers and generates a list that contains all the same numbers, except that the odd numbers are now replaced with ‘x’.

The syntax changes if you use an **if-else**:

[f(x) if condition else g(x) for x in some_iterable]

Red – Output, Purple – Conditional, Blue – Original sequence

```
replaced = ???
```

Dictionary comprehension

Exactly the same as a list comprehension, but our output expression is now a key - value pair.

Eg: Double each value in a dictionary:

my_dictionary = {'a': 1, 'b': 2, 'c':3, 'd':4}

dictionary_2 = {k : v*2 for key,value in my_dictionary.items()}

Output: {'a': 2, 'b': 4, 'c' :6, 'd':8}

{ f(key): g(value) for key,value in some_iterable if conditional }

Red – Output, Blue – Original iterable, Purple - condition

Dictionary comprehension

We can even make dictionaries from lists, ranges and other iterables!

```
{ chr(i + 65) : i + 65 for i in range(0,26)}
```

Red – Output, Blue – Original iterable, Purple – condition (optional)

How we might read the expression is:

Create a key, value pair in the range 0-26 and store it in a dictionary **{}**

{ 'A': 65, 'B':66, 'C':67, ….. , 'Z':90 }

Why should we use comprehensions?

What's the point of Comprehensions?

- Avoid messy while and for loops. Which ones looks easier to you?

```
#using for loop
list = []
for x in range(0,10):
    if x % 2 == 0:
        list.append(x)
    else:
        list.append('x')


#same result using list comprehension
list = [x if x % 2 == 0 else 'x' for x in range(0,10)]
```

What’s the point of Comprehensions?

- Comprehensions hide the working of iterators
- Comprehensions leverage the power of iterators to do complex operations.
- Comprehensions are faster
- Comprehensions are readable
- Comprehensions are maintainable

Comprehensions with more complex data.

Let's see an example where we parse the following log file snipper with Comprehensions.

```
user@host> file show /var/log/processes Feb 22 08:58:24 router1 snmpd[359]:
%DAEMON-3-SNMPD_TRAP_WARM_START: trap_generate_warm: SNMP trap: [Warning]
warm start
Feb 22 20:35:07 router1 snmpd[359]:
%DAEMON-6-SNMPD_THROTTLE_QUEUE_DRAINED: [Warning]
trap_throttle_timer_handler: cleared all throttled traps
Feb 23 07:34:56 router1 snmpd[359]:
%DAEMON-3-SNMPD_TRAP_WARM_START: trap_generate_warm: SNMP trap: warm start
Feb 23 07:38:19 router1 snmpd[359]:
%DAEMON-2-SNMPD_TRAP_COLD_START: trap_generate_cold: [Warning] SNMP trap:
cold start
```

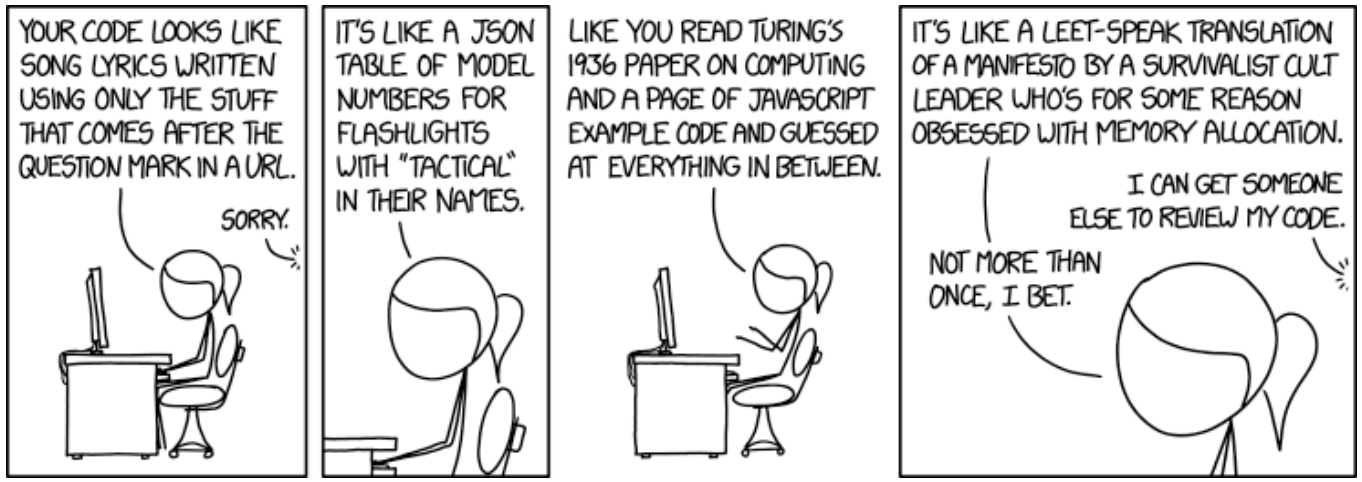
file_comprehension.py

Abusing Comprehensions

Do not write comprehensions, if the expression is so convoluted that it's not readable and maintainable.

Our goal as developers is to write maintainable code.

Unless we are saving a lot of space and gaining tons of speed, it's ok to switch to a for loop or while loop **if** it is more readable

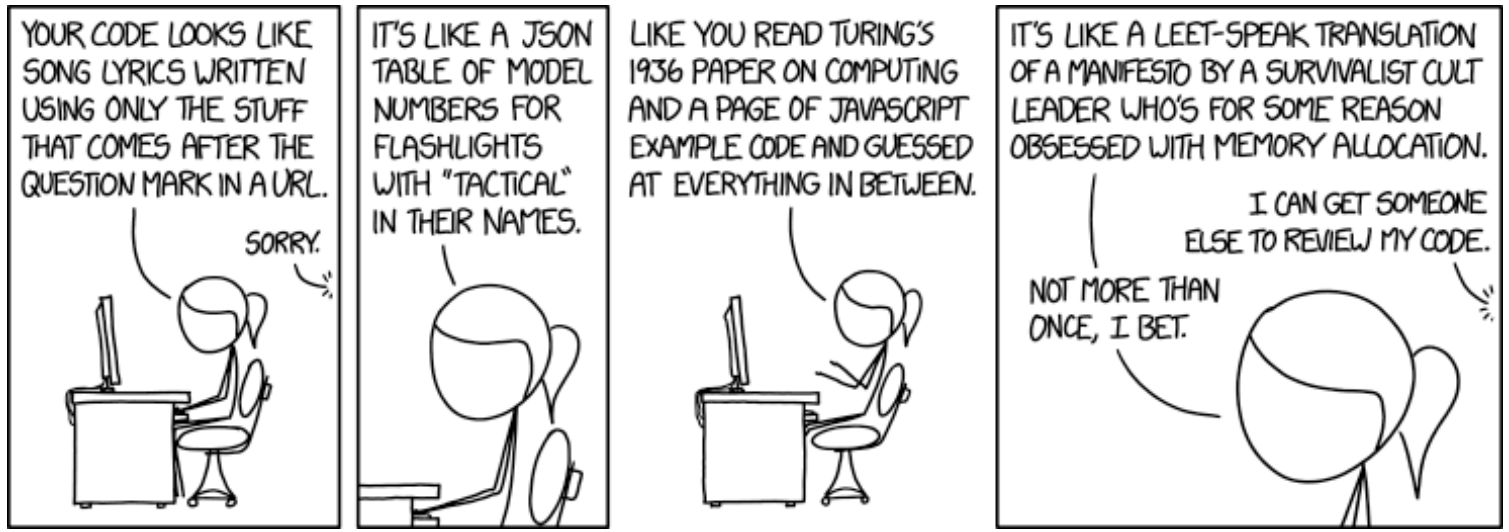


Abusing Comprehensions

That being said comprehensions are the preferred pythonic way of operating oniterables.

And there is the fact that **programmers like writing code that's concise and feels magical.**

(Except for the person debugging your code)



Abusing Comprehensions

And if the previous reasons weren't enough, if you do the following in python:

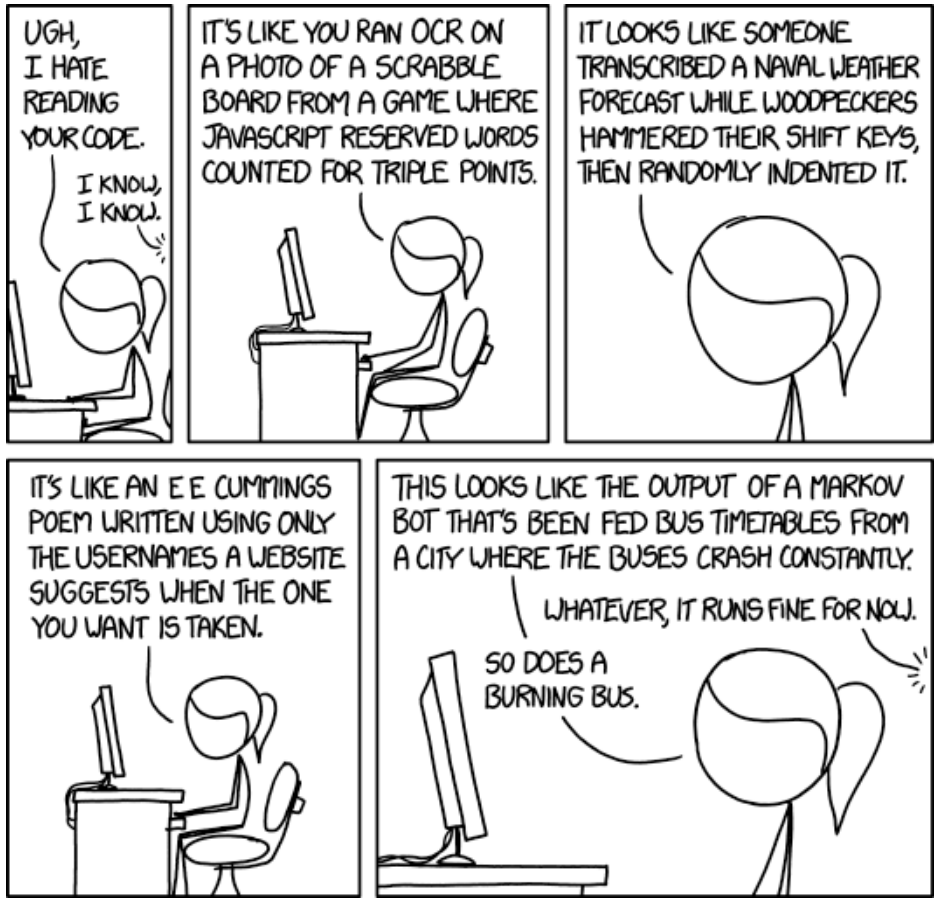
- Use tons of for loops to carry out operations on iterables

- Try using for loops traditionally with range(num_iteration)

- Don't use protocols

- Put each class in its own module all the time.

Then you **might** get a reaction like this from other python developers.



Functions as Objects

Functions are objects

Try typing the following in your command shell/Jupyter or PyCharm:

```
>>> def some_function():
...     print(“Something”)


>>> function_obj = some_function
>>> print(function_obj)
>>> print(function_obj.__name__)
>>> print(function_obj.__class__)
>>> print(type(function_obj))
>>> function_obj()
```

Functions are objects

Try typing the following in your command shell/Jupyter or PyCharm:

The variable `function_obj` is pointing to the function `some_function.`

Can now use the variable `function_obj` as though it was the function itself

```
>>> def some_function():
...     print(“Something”)


>>> function_obj = some_function
>>> print(function_obj) #<function some_function at 0x007B67C0>
>>> print(function_obj.__name__) #some_function
>>> print(function_obj.__class__) #<class 'function'>
>>> print(type(function_obj)) #<class 'function'>
>>> function_obj() #Something
```

Dictionaries and Functions

Functions are treated in an OOP manner, that is to say, they are objects.

Everyone has already been using functions as objects!

- Some of you had errors in your code where you forgot to mention the parenthesis () with a function name but the compiler complained as if something was wrong with an object
- Some of you have tried putting functions in dictionaries and mapped them to menu choices.

Let's look at some code to refresh out memory

function_objects.py

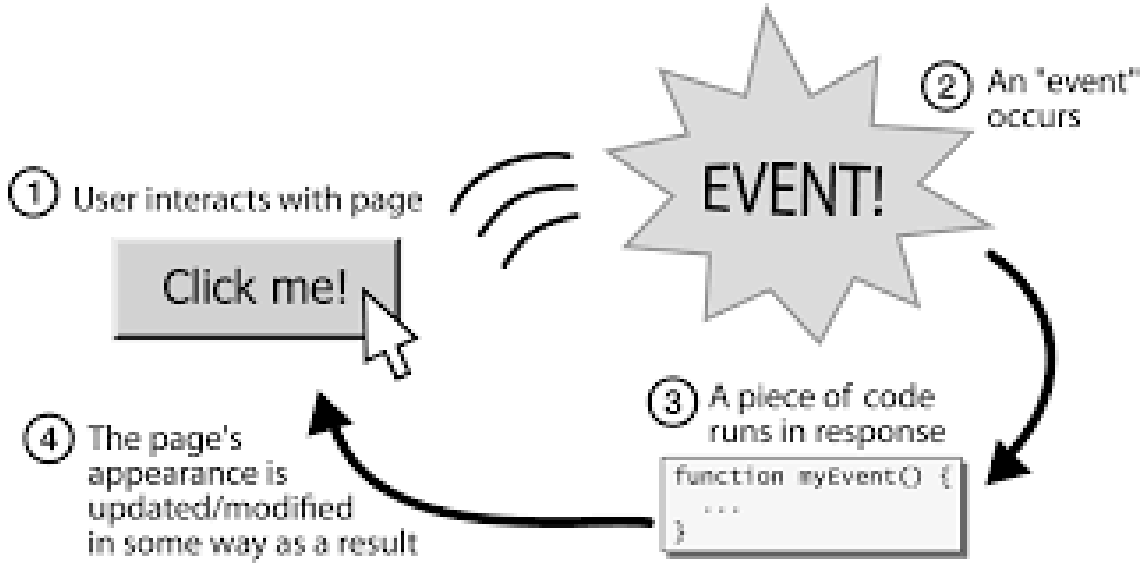
```
input_map = {
    1: sum,
    2: subtract,
    3: multiply,
    4: divide,
}
```

Functions as Objects: **Event Driven Programming**

1. We can store functions as objects
2. Pass them around via arguments
3. Call them later in the future when some condition has been met.

These functions that are passed to be called back later are known as **callbacks**.

That is the essence of event-driven programming.

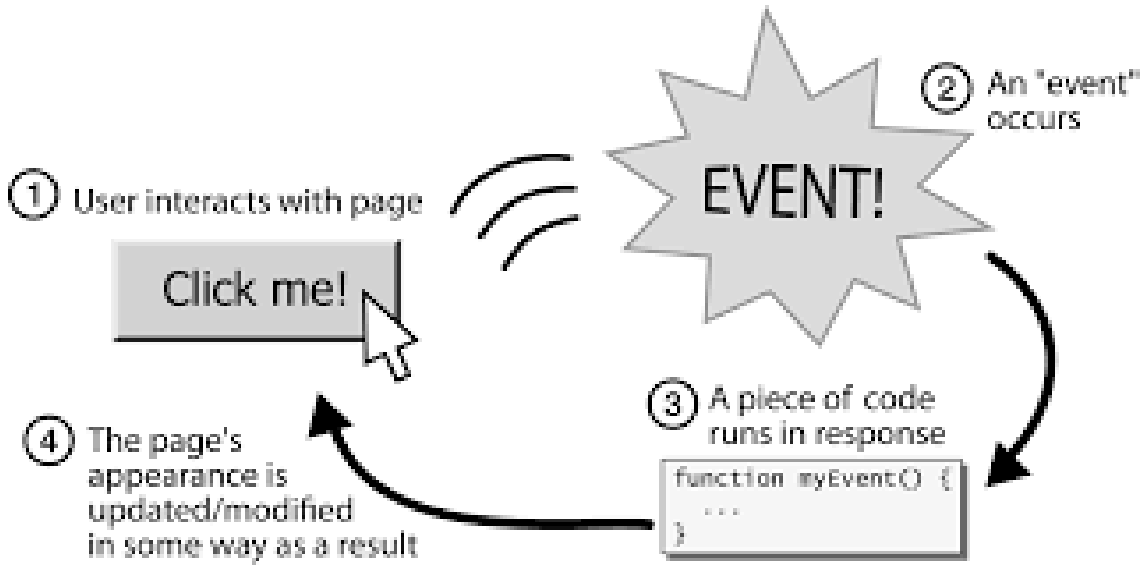


Functions as Objects: **Event Driven Programming**

Functions that are meant to handle an event such as a mouse or button click are known as **Event Handlers**

Event Handlers are just another type of callback. You could fire multiple event handlers when an event occurs.

That is the essence of event-driven programming.



Event Driven Programming

This is extremely simple in Python.

We don’t need to worry about advanced constructs in other languages like delegates

We don’t need an artificial object to wrap around 1 function so we can pass it around.

Things are much simpler in Python since functions are objects themselves!

Let’s look at an example!

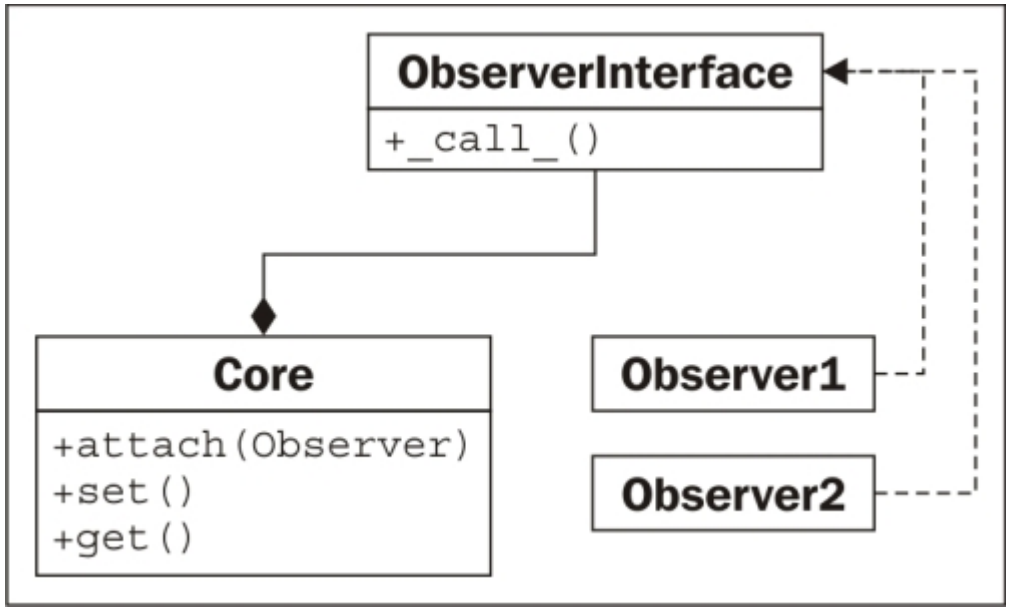
(refer to the callback_example.py sample code)

First design pattern!
The Observer Pattern

The observer pattern is honestly just a formal way of organizing the code that we just saw.

You have a number of **observers** that are observing *something*. This could be a timer, a button, an inventory system, anything. This is known as the **Core**.

When **something changes**, or some **significant state has been reached** (such as the timer finished counting down) then the Core notifies its observers by calling all the callbacks that were registered by them

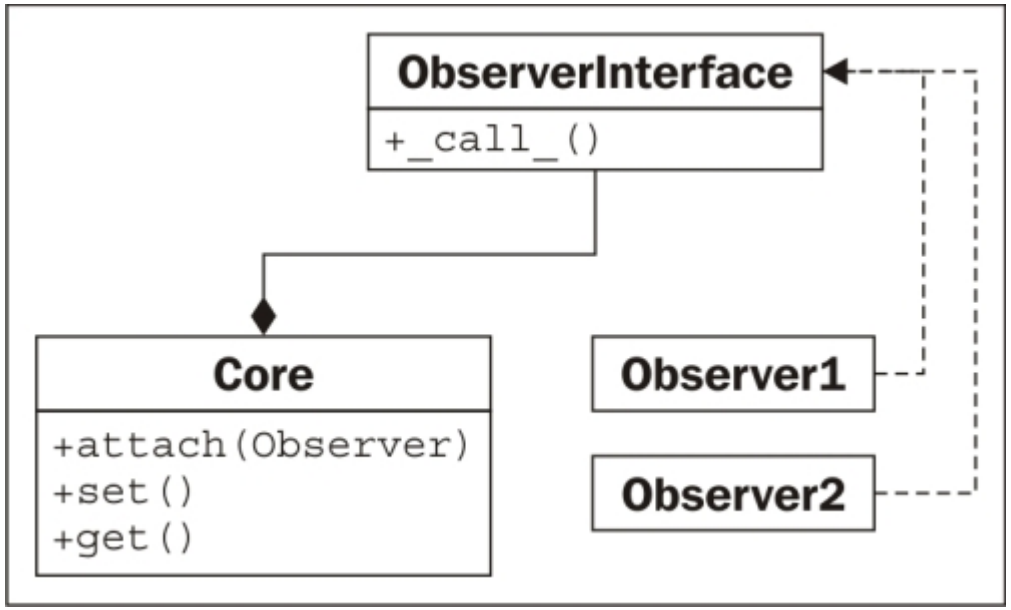


First design pattern!
The Observer Pattern

These are the requirements:

1. All the callback functions **accept the same parameters**. That is, all the observers implement either the same interface, or a similar method (informal interface)
2. The core doesn't care who's observing it or what they do when a change occurs.
3. All the core cares about is the fact that **it has to run a set number of callbacks** that have the same interface to notify *some* observers that do *something*.

This is a great way of de-coupling code in GUI's.



observer_pattern.*py*

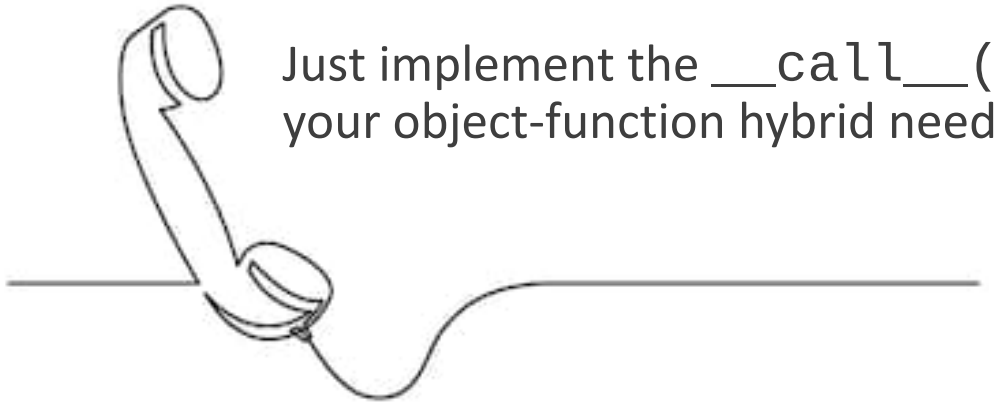
Callable Objects

We have one last topic before we wrap up!

We know functions can be treated as objects.

What if we wanted to treat objects as functions?

We can totally do that!



Just implement the `__call__(self)` method and add any parameters your object-function hybrid needs. *callable_object.py*

Summary

- Iterators are important. They allow for powerful constructs such as comprehensions.
- Python programmers are picky about how code is written
- Comprehensions are your friends. Practice writing them.
- Functions can be objects too, don’t discriminate
- Objects can be function too, don’t discriminate
- Also, the observer pattern is one of the most important design patterns you will ever encounter in any language. Pay attention to it!

That’s it for this week!

Midterm coming up soon

We’ll do some review next Friday

Submit questions to: https://forms.gle/oWdXjJb4nWhJ9fpr8

Can submit multiple questions, but be specific