Functional Programing

May 10, 2022

You have probably heard this term a lot:

Object Oriented Programming

O.O.P. is a Paradigm

or a “way or programming”

Object Oriented Programming

The programmer defines a series of objects, and control flow is defined as a series of operations on these objects.

There are other paradigms

- **Imperative**
- **Declarative**
- **Structured**
- **Procedural**
- **Functional**
- **Function-Level**
- **Object-Oriented**
- **Flow-Driven**
- **Logic**
- **Constraint**
- **Aspect-Oriented**
- **Reflective**
- **Array**

There are other paradigms

- **Imperative**
- **Declarative**
- **Structured**
- **Procedural**
- **Functional**
- **Function-Level**
- **Object-Oriented**
- **Flow-Driven**
- **Logic**
- **Constraint**
- **Aspect-Oriented**
- **Reflective**
- **Array**

We often combine these naturally,
and use many at once

There are other paradigms

- **Imperative**
- **Declarative**
- **Structured**
- **Procedural**
- **Functional**
- **Function-Level**
- **Object-Oriented**
- **Flow-Driven**
- **Logic**
- **Constraint**
- **Aspect-Oriented**
- **Reflective**
- **Array**

We often combine these naturally,
and use many at once

Functional Programming

The Functional paradigm is based on the philosophy of expressing program control flow through a series of (often nested) function calls, rather than through loops, variables, or classes.

Functional Programming

The Functional paradigm is based on the philosophy of expressing program control flow through a series of (often nested) function calls, rather than through loops, variables, or classes.

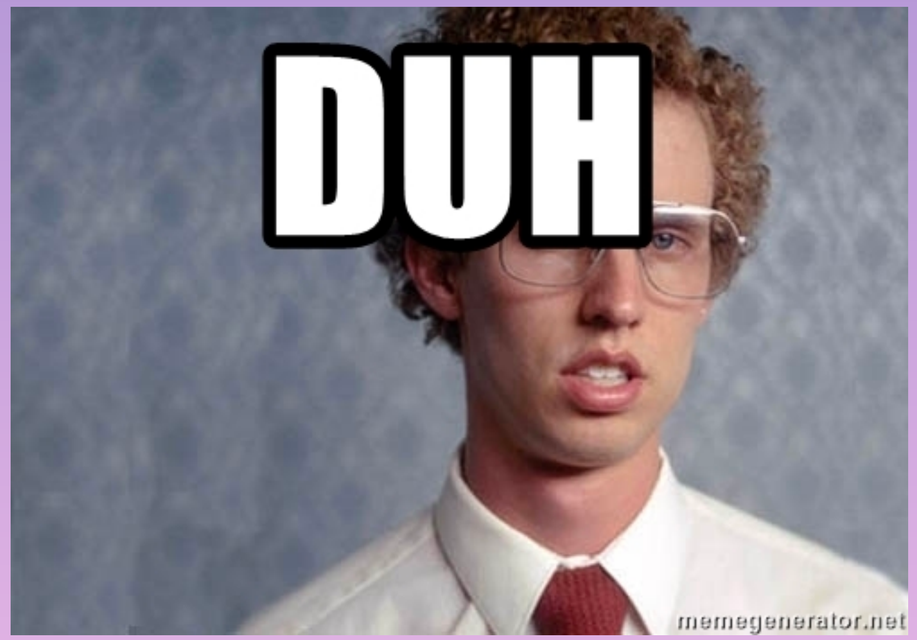
Python has a lot of features that allow for functional programming!

Code Comparisons: Multiple Paradigms

```
def squared_divisible_by_2_9(lst):
    ret_lst = []
    for elem in lst:
        if elem**2 % 2 == 0 and elem**2 % 9 == 0:
            ret_lst.append(elem**3)
    return ret_lst
```

Code Comparisons: Functional Programming

```
def squared_divisible_by_2_9(lst):
    return list(map(
        lambda x: x**3,
        filter(
            lambda x: x**2 % 2 == 0 and x**2 % 9 == 0,
            lst
        )
    ))
```


DUH
memegenerator.net

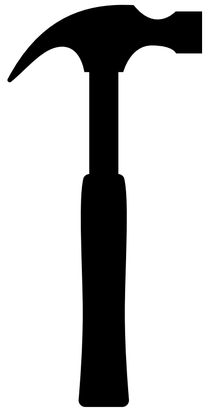
Code Comparisons

```
def squared_divisible_by_2_9(lst):
    return list(map(
        lambda x: x**3,
        filter(
            lambda x: x**2 % 2 == 0 and x**2 % 9 == 0,
            lst
        )
    ))
```

Python has many features that allow for Functional Programming

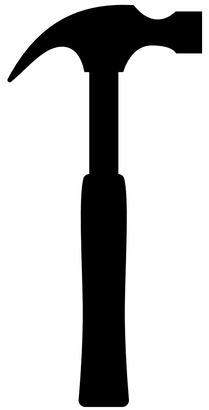
We already have some of these tools

We already have some of these tools



Functions

We already have some of these tools



Functions

Functions as
Parameters and
Return Values

Recap of Decorators

```
def print_arguments(fn):
    def fn_prime(*args, **kwargs):
        print("Arguments: {}, {}".format(args, kwargs))
        fn(*args, **kwargs)
    return fn_prime
```

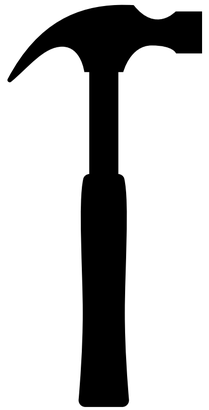
Recap of Decorators

```
def foo(a, b, c=1):
    print((a + b) * c)

foo = print_arguments(foo)
foo(2, 1, c=3)

# Arguments: (2, 1) {'c': 3}
# 9
```

Recap of Decorators

```
def print_arguments(fn):
    def fn_prime(*args, **kwargs):
        print("Arguments: {}, {}".format(args, kwargs))
        fn(*args, **kwargs)
    return fn_prime

@print_arguments
def foo(a, b, c=1):
   print((a + b) * c)

foo(2,1,c=3)

# Arguments: (2, 1) {'c': 3}
# 9
```

We already have some of these tools

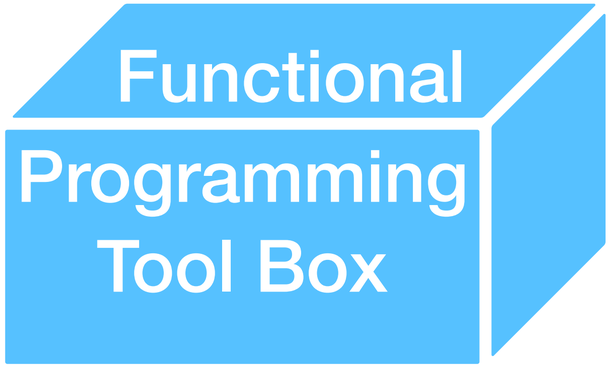


Functions

Functions as
Parameters and
Return Values

We already have some of these tools

But we need more to get to this code

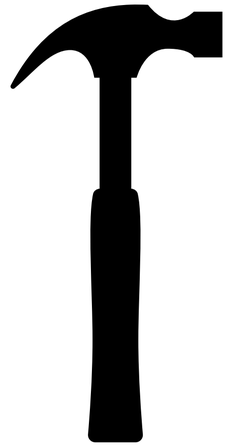



```
def squared_divisible_by_2_9(lst):
    return list(map(
        lambda x: x**3,
        filter(
            lambda x: x**2 % 2 == 0 and x**2 % 9 == 0,
            lst
        )
    ))
```

Functional Programming Basics

New Tool!

New Tool!



Iterators

Iterables

```
lst = ["Parth", "Unicorns", "Tara", "Horses"]
tup = ("CS41", "BIO101")
dctnry = {"CS": [41,106], "BIO": [82, 101, 190]}
```

Collections of objects that we can iterate over. We can use “for” or “while” loops through the contents and obtain something useful.

Iterators

```
lst = ["Parth", "Unicorns", "Tara", "Horses"]
tup = ("CS41", "BIO101")
dctnry = {"CS": [41, 106], "BIO": [82, 101, 190]}

lst_iter = lst.__iter__()
tup_iter = tup.__iter__()
dctnry_iter = dctnry.__iter__()
```

Iterators

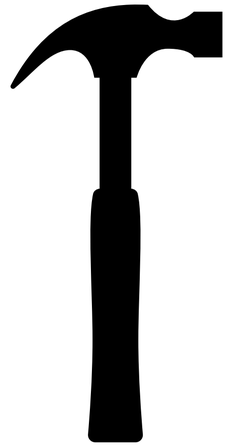
```
lst = ["Parth", "Unicorns", "Tara", "Horses"]

#functionally same code
for elem in lst:
    print(elem)


#functionally same code
lst_iter = lst.__iter__()
while True:
    try:
        print(lst_iter.__next__())
    except StopIteration:
        break
```

Why is this useful?

Allows us to get next item in a data set one at a time, and only when we need it

Allows for us to access iterable data outside of being in a loop

New Tool!



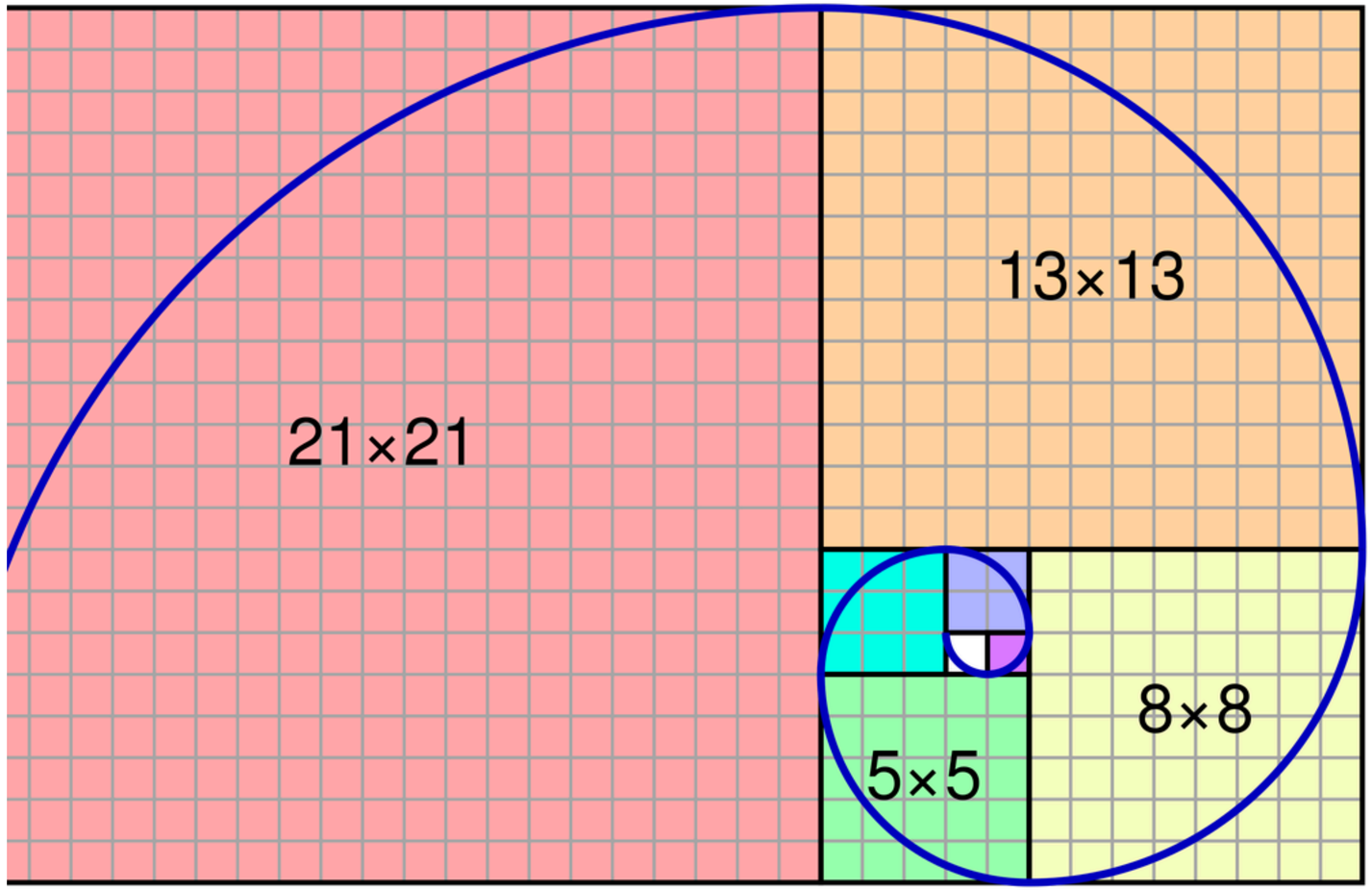
Iterators

New Tool!

Iterators

New Tool!

Let’s use this to deal with some infinite data



Fibonacci Sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987 \ldots$$

Each number is the sum of the previous two numbers.

How could you write code that generates this sequence and then iterates through it?

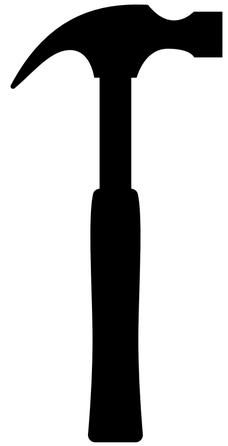
$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987 \ldots$$

Each number is the sum of the previous two numbers.

Fib Code

New Tool!

New Tool!



Generators

Generators Introduce “Yield”

-When next is called on a generator, it runs until a yield statement is reached, at which point the value referenced by the yield will be returned to the caller, and execution of the generator will pause.

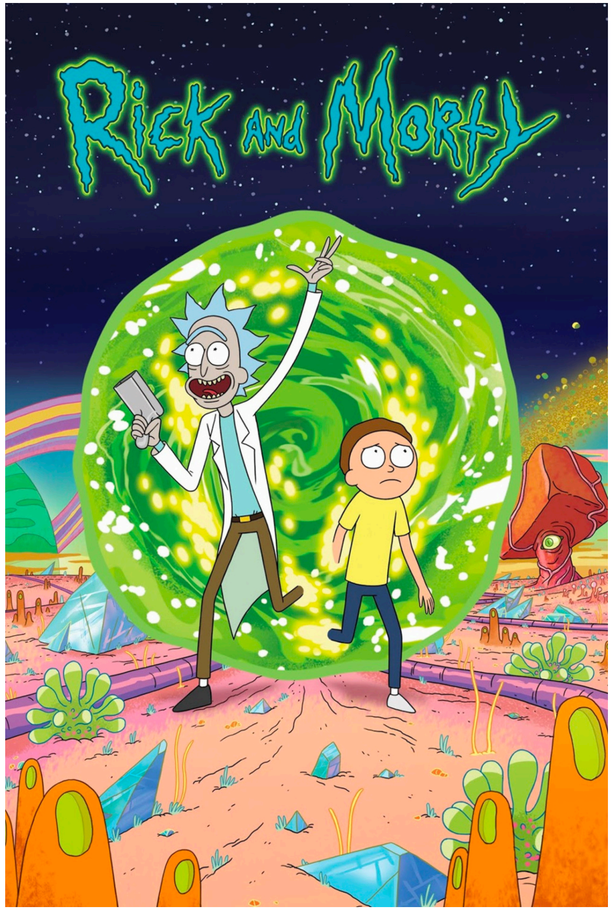
-The next time that the __next__() method is called, execution of the generator will resume, until a yield statement is reached - at that point, again, execution will pause and the value referenced by the yield will be returned to the caller

Generator Fib Code

Generators

```
#functionally the same as
for elem in generator:
    print(elem)

#functionally the same as
while True:
    try:
        print(generator.__next__())
    except StopIteration:
        break
```

Generators API mix!!



Generators API mix!!

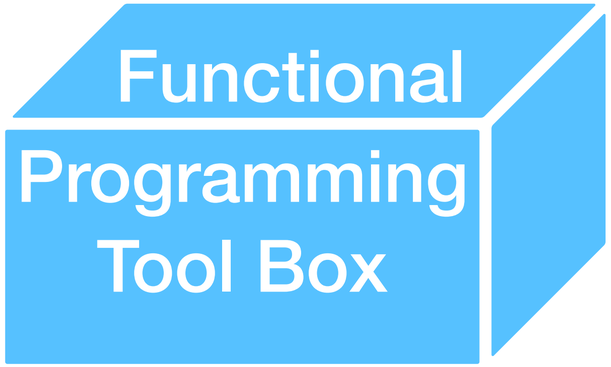
Your goal:

-Write code that goes through characters in order of IDS

-Gives them one at a time based on user input

-Uses a generator

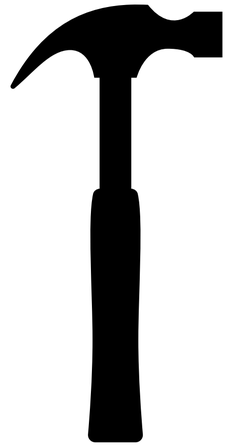
Now let’s get back to the terms seen here




```
def squared_divisible_by_2_9(lst):
    return list(map(
        lambda x: x**3,
        filter(
            lambda x: x**2 % 2 == 0 and x**2 % 9 == 0,
            lst
        )
    ))
```

New Tool!

New Tool!



Lambda

Lambda: Anonymous Inline Functions

```
lambda params : expression
```

Let’s see how these are used

```
def square_add_two(x,y):
    return  x**2 + y**2
```

Let’s see how these are used

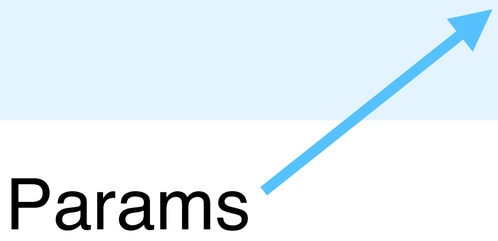
```
def square_add_two(x,y):
    return  x**2 + y**2
```

```
lambda x, y : x**2 + y**2
```

Let’s see how these are used

```
def square_add_two(x,y):
    return  x**2 + y**2
```

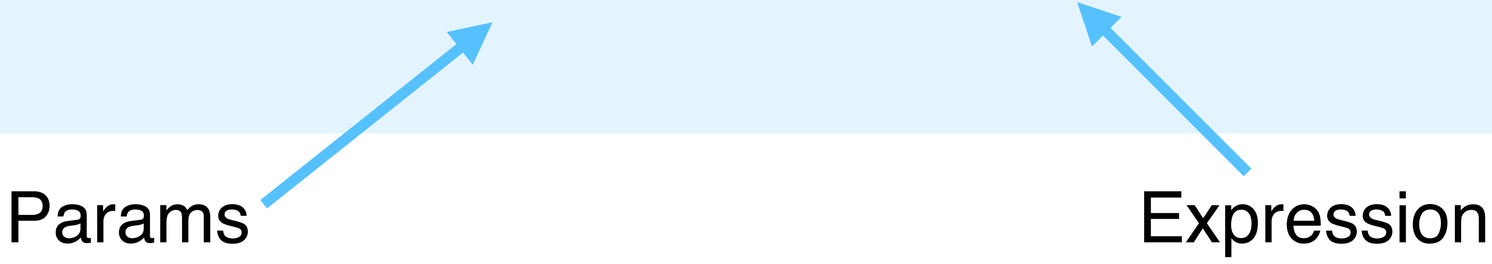
```
lambda x, y : x**2 + y**2
```



Let’s see how these are used

```
def square_add_two(x,y):
    return  x**2 + y**2
```

```
lambda x, y : x**2 + y**2
```



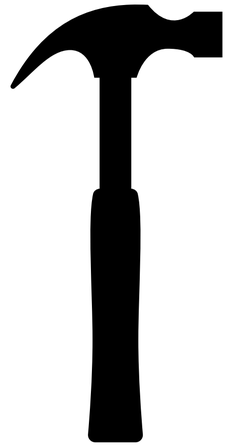
To call a lambda, you can put the arguments in parenthesis after the definition (also in paraenthesis)

```
(lambda x, y : x**2 + y**2)(8,5)
```

How was this used in our example code?

```
def squared_divisible_by_2_9(lst):
    return list(map(
        lambda x: x**3,
        filter(
            lambda x: x**2 % 2 == 0 and x**2 % 9 == 0,
            lst
        )
    ))
```

Add that to your tools!



Lambda

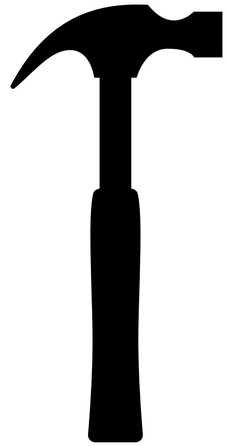
Add that to your tools!

Lambda

Add that to your tools!

New Tools!

New Tools!



Map and
Filter

Map and Filter

These allow you to change arrays in streamline way

Map and Filter

These allow you to change arrays in streamline way

```
map(fn, iterable)
```

In Python, the map function takes in two parameters, a function and an iterable, and it applies each the function to each element of that iterable.

Map in Action

```
lst = [1, 2, 3]
map(lambda x : x**2, lst)
# => <map object at 0x104885310>
```

By why this map object?

```
map_obj = map(lambda x : x**2, lst)
map_obj.__next__()                      # => 1
map_obj.__next__()                      # => 4
map_obj.__next__()                      # => 9
map_obj.__next__()                      # => StopIteration
```

Map can take in finite (list, tuple, ect) or infinite (generator) objects. Because infinite and finite objects can be represented by generators, it returns a map object, which acts as a generator.

But tbh this is not always helpful

```
map_obj = map(lambda x : x**2, lst)
list(map_obj)                                  # => [1, 4, 9]
```

We can use the list(), tup(), constructors to turn this map object into our input type

Map and Filter

These allow you to change arrays in streamline way

Map and Filter

These allow you to change arrays in streamline way

```
filter(predicate, iterable)
```

The filter function behaves similarly to the map function: it takes in two arguments, a predicate, and an iterable, and it returns only the elements of the iterable for which the predicate is True.

How are these used in our example code?

```
def squared_divisible_by_2_9(lst):
    return list(map(
        lambda x: x**3,
        filter(
            lambda x: x**2 % 2 == 0 and x**2 % 9 == 0,
            lst
        )
    ))
```

Let’s try out map and filter

[‘3600’, ’41’, ’84’]	[3600, 41, 84]
[‘Tara’, ‘Parth’, ‘Theo’]	[4,5,4]
[‘CS41’, ‘CS106A’, ‘BIO84’, ‘CHEM18’, ‘CS161’]	[‘CS41’, ‘CS106A’, ‘CS161’]