F(unctional_programming)

January 27, 2020

Week 4 of CS 41

Today in CS41:

- What is functional programming and why do I care?
- `lambda`
- `map` and `filter`
- Iterators/Generators
- Decorators

Peak weirdness **will be achieved at the end of lecture!**

What *Really* are Functions?

First-Class Functions

```python
def echo(arg):
    return arg

type(echo) # => <class 'function'>
id(echo) # => 4393732128
print(echo) # => <function echo at 0x105e30820>

foo = echo
type(foo) # => <class 'function'>
id(foo) # => 4393732128
print(foo) # => <function echo at 0x105e30820>

isinstance(echo, object) # => True
```

Functions are Objects!

What is Functional Programming?

Functional Programming – Overview

- Programming Paradigms
  - **Procedural** – program is a sequence of instructions that tell the computer what to do. Examples: C, Pascal, Unix shell.
  - **Object-Oriented** – collections of objects which maintain internal state and support querying and changing of the internal state. Examples: Java, Smalltalk.
  - **Declarative** – describe the problem to be solved, language implementation figures out the details. Examples: SQL, Prolog.
  - **Functional** – programs decompose into sets of functions, each of which takes inputs and produces outputs without internal state. Examples: Haskell, OCaml.

Python is a Multi-Paradigm Language

Supports many paradigms – programming is a choose-your-own adventure!

Functional Programming – Example

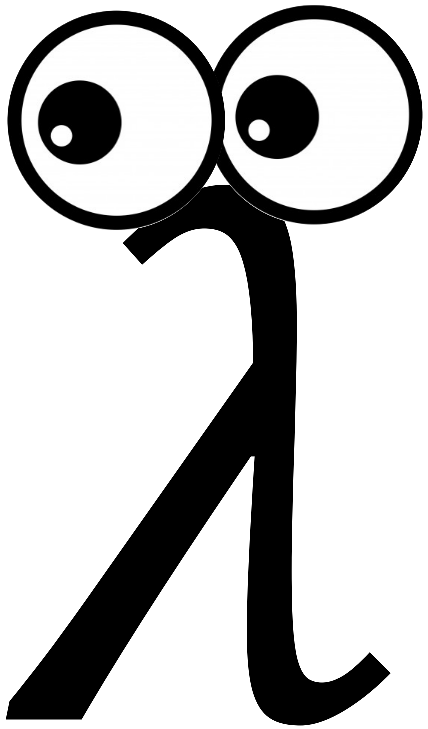
```python
# Procedural - "program flow"
def get_odds(arr):
    ret_list = []
    for elem in arr:
        if elem % 2 == 1:
            ret_list.append(elem)
    return ret_list

# Functional - "programs are sets of functions"
def get_odds(arr):
    return list(filter(lambda elem: elem % 2 == 0, arr))
```

Why Functional Programming?

- **Simplify debugging** - line-by-line invariants, so easier to find points of failure.
  - Fewer variables designed exclusively to, say, track an index.
- **Shorter, cleaner code** – recall the earlier example!
- **Modular** - functions are often reusable, so it's faster to code the next thing. Also enables module-by-module testing and debugging.

Lambdas

Smaller, cuter functions!



Lambdas: Inline, Anonymous Functions

- Anonymous, on-the-fly functions, which can be passed as parameters into other functions.

```
>>> lambda params: expression
```

```
>>> # Check whether the first item in a pair (tuple) is greater than the
>>> # second
>>>
>>> lambda tup: tup[0] > tup[1]
```

Lambdas: Inline, Anonymous Functions

- Lambdas can customize the functionality of Python functions!

```
>>> # Find the maximum in a list of pairs by value of the second element.
>>>
>>> pairs = [(3, 2), (-1, 5), (4, 4)]
>>> max(pairs, key=lambda tup: tup[1])
(-1, 5)
```

Lambdas: Inline, Anonymous Functions

- Lambdas can customize the functionality of Python functions!

```
>>> # Sort a collection of pairs by the value of the second element.
>>>
>>> pairs = [(3, 2), (-1, 5), (4, 4)]
>>> sorted(pairs, key=lambda tup: tup[1])
[(3, 2), (4, 4), (-1, 5)]
```

Lambdas: Inline, Anonymous Functions

- Lambdas can customize the functionality of Python functions!
- This, though syntactically valid, is bad. Why?

```
>>> trip = lambda x: 3*x
```

- The whole point of a lambda is to be used inside a function call, then discarded. If we are binding it to a name to be called in the future, it may as well just be defined as a function!

The `map` Function

X Marks the Spot!

A Common Pattern

- Applying a function elementwise to an array, storing the result.

```
def length_of_all_elements(arr):
    ret_arr = []
    for elem in arr:
        ret_arr.append(len(elem))
    return ret_arr



>>> length_of_all_elements(["Parth", "Unicorn", "Michael"])
[5, 7, 7]
```

List Comprehensions to the Rescue!

- We saw this earlier – list comprehensions solve our problem!

```
>>> [f(x) for x in iterable]
```

```
def length_of_all_elements(arr):
    return [len(elem) for elem in arr]


>>> length_of_all_elements(["Parth", "Unicorn", "Michael"])
[5, 7, 7]
```

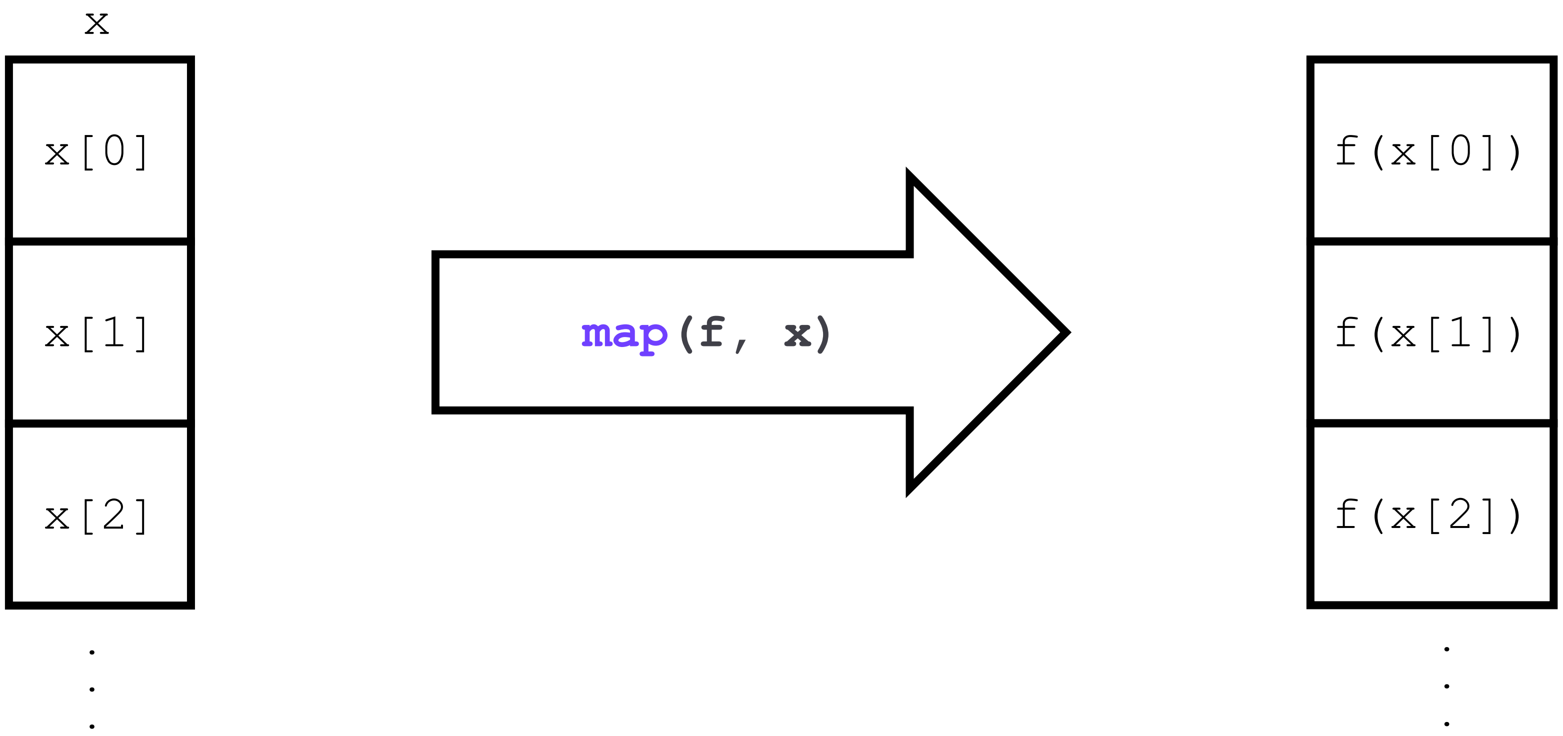
Does Anybody Have a `map`?

- Maps a function onto an iterable, returning another iterable.
- No mention of the elements within the iterable (unlike a comprehension).

```
>>> map(f, iterable)
```

Does Anybody Have a `map`?

- Maps a function onto an iterable, returning another iterable.
- No mention of the elements within the iterable (unlike a comprehension).



Does Anybody Have a `map`?

- Maps a function onto an iterable, returning another iterable.
- No mention of the elements within the iterable (unlike a comprehension).

```
>>> map(f, iterable)
```

```
def length_of_all_elements(arr):
 return list(map(len, arr))

>>> length_of_all_elements(["Parth", "Unicorn", "Michael"])
[5, 7, 7]
```

The `filter` Function

#NoFilter #JokesDefinitelyAFilter

Another Common Pattern

- Extracting elements of an iterable which fulfill certain criteria.

```
def starts_with_m(arr):
    ret_arr = []
    for elem in arr:
        if elem[0].lower() == "m":
            ret_arr.append(elem)
    return ret_arr


>>> starts_with_m(["Michael", "Parth"])
["Michael"]
```

List Comprehensions – Take Two!

- As we saw earlier, we can use a list comprehension to construct a list, filtered by a conditional.

```
>>> [x for x in iterable if condition]
```

```
def starts_with_m(arr):
    return [x for x in arr if x[0].lower() == "m"]


>>> starts_with_m(["Michael", "Parth"])
["Michael"]
```

(Not an Instagram) `filter`

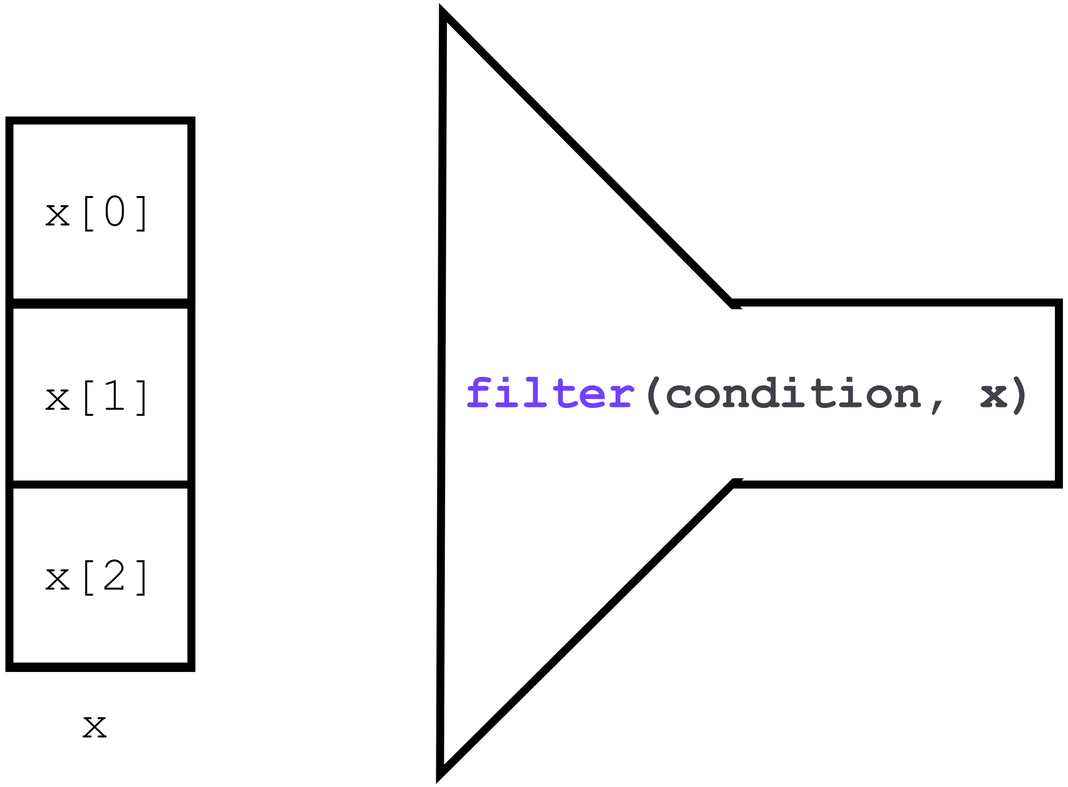
- Generates an iterable by filtering the elements of another iterable based on a provided condition. [1]

```
>>> filter(condition, iterable)
```

[1] Here, "condition" is a function that takes in an element of the iterable and returns something truthy or falsy.

(Not an Instagram) `filter`

- Generates an iterable by filtering the elements of another iterable based on a provided condition.



(Not an Instagram) `filter`

- Generates an iterable by filtering the elements of another iterable based on a provided condition. [1]

```
>>> filter(condition, iterable)
```

```
def starts_with_M(arr):
    return list(filter(lambda word: word[0].lower() == "m", arr))


>>> starts_with_M(["Michael", "Parth"])
["Michael"]
```

[1]Here, "condition" is a function that takes in an element of the iterable and returns something truthy or falsy.

Runtime and Space Considerations

- Memory
  - List comprehensions buffer all computed results.
  - Map/Filter compute elements only when called (more memory efficient!)
- Speed
  - List comprehensions don't have function call overhead. (The call to **`map`** or **`filter`** comes with extra overhead if you pass a lambda into it).
  - Filter/Map are occasionally faster, but the function call overhead usually means they are not.

Attendance Form

Link: http://iamhere.stanfordpython.com

Code: **FUNctional programming**

If you're not here, fill out this alternative form:

https://bit.ly/2O4Gpyr

Logistics

Assignment 0	Grades will be released later today – great job everyone!
Assignment 1	Due February 4, 2020, at 11:59PM.
Labs	Lab on Wednesday – bring a charged computer!

A0 – Common Errors

- Python file components:
  - `#!/usr/bin/env python3`
  - `main()` functions, and `if __name__ == "__main__"`
- Comments – just because this class isn't CS106B doesn't mean you shouldn't comment your code!
- Context managers!
  - `with open("answers.txt") as f:`
  - Safety first – if the program breaks during file reading, context managers safely exit.
- Using `exit()` to end the program.
  - If imported as a module, this also exits the caller!

Iterators

An Old Example...

- Let's revisit filtering out strings in a list that start with "M".

```
def starts_with_M(arr):
    return list(filter(lambda st: st[0].lower() == "m", arr))
```

Why the conversion to `list`?

What’s Your Type?

- What do map and filter actually return?

```
>>> arr = ["Parth", "Michael"]
>>> map_to_investigate = map(lambda x: x + "likes unicorns", arr)
>>> map_to_investigate
<map object at 0x10c6a0550>
>>> filter_to_investigate = filter(lambda x: x == "Unicorn", arr)
>>> filter_to_investigate
<filter object at 0x10c6a0510>
```

Iterators

- **`map`** and **`filter`** objects are iterators: represent a finite or infinite stream of data.
- Use the **`next(iterator)`** function to iterate through the elements of an iterator.
  - Raises **`StopIteration`** error upon termination.
- Use **`iter(data_structure)`** to build an iterator over a data structure.
  - E.g. **`iter([1, 2, 3])`** builds an iterator over a list

Example - Iterators

```
>>> names = ["Parth", "Michael", "Unicorn"]
>>> length_filter = filter(lambda word: len(word) >= 7, names)
>>> next(length_filter)
"Michael"
>>> next(length_filter)
"Unicorn"
>>> next(length_filter)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
StopIteration
>>>
```

For Loops Use Iterators

```python
# This code...
for data in data_source:
    do_something_to(data)

# ...is equivalent to...
data_iter = iter(data_source)
while True:
    try:
        data = next(data_iter)
    except StopIteration:
        break
    else:
        do_something_to(data)
```

Reading Iterators

- Finite iterators can be read into data structures.

```
def starts_with_M(arr):
    return list(filter(lambda st: st[0].lower() == "m", arr))
```

Taking the output from an iterator, creating a `list` from it!

Generators

“Lazy List Comprehensions”

“Resumable Functions”

- Ordinary functions
  - Return a single, computed value
  - Each call generates a new local namespace and new local variables.
  - Namespace is discarded upon exit.
- Generators
  - Return an iterator that will generate a stream of values
  - Local variables aren’t discarded upon suspension – pick up where you left off!

The Fibonacci Sequence

```
# Let's write a generator to generate the Fibonacci sequence!
def fib():
    a, b = 0, 1
    while True:
        a, b = b, a+b
        yield a


>>> g = fib() # Namespace created, fib() pushed to stack.
>>> type(g)
<class 'generator'>
>>> next(g)
1
>>> next(g)
1              # next(g) : 2, 3, 5, 8, 13, 21, 34...
>>> max(g)       # What happens?
```

Lazy Generation

- We can use our generator to represent infinite streams of data in a finite way.
  - Since we can't work with the whole Fibonacci sequence – it's infinite – generators let us perform computation on elements of the sequence as we need to.

```
# Generating the Fibonacci sequence only on demand
def fibs_under(n):
    for num in fib(): # The generator we just made! Loops over 1, 1, 2...
        if num > n:
            break
        print(num)
```

Why Use Generators?

- Compute data on demand
  - Avoids expensive function calls
  - Reduces memory buffering
- Allows us to define infinite streams of data
  - We couldn’t do this before!

Decorators

A Tale of Two Paradigms

Functions as Arguments

- We've already seen functions as arguments before!

```
# We saw this in map and filter!
map(fn, iterable)
filter(fn, iterable)


# We can also write our own functions which take in functions as arguments
def do_twice(fn, *args):
    fn(*args)
    fn(*args)


>>> do_twice(print, "Parth is a wonderful person")
Parth is a wonderful person
Parth is a wonderful person
```

Functions as Return Values

- We can also return functions from functions!

```
def make_divisibility_test(n):
    def is_divisible_by(m):
        return m % n == 0
    return is_divisible_by


>>> div_test = make_divisibility_test(5)    # "test" is a function!
>>> div_test(256)
False
>>> div_test(10)
True
>>>
```


Is it too much to ask for both?

Decorators: Best of Both Worlds

- Decorators take in a function, modify the function, then return the modified version.
- Our first decorator: modifies a function by printing out the arguments passed into it.

```
# Our first decorator
def debug(function):
    def modified_function(*args, **kwargs):
        print("Arguments:", args, kwargs)
        return function(*args, **kwargs)
    return modified_function
```

What in the world is this?
Pause for questions!

Using our Decorator

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


>>> foo = debug(foo)
>>> foo(2, 3)
Arguments: (2, 3) {}            # Printed from the debugging decorator
5                               # Returned from the function
>>> foo(2, 1, c=3)
Arguments: (2, 3) {'c': 1}      # Printed from the debugging decorator
9                               # Returned from the function
>>>
```

Making Things More Pythonic

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

>>> foo = debug(foo)
```

This isn't cool...

This method of applying a decorator forces us to overwrite the namespace binding for "foo" in the global scope. Yuck!

Making Things More Pythonic

```
@debug
def foo(a, b, c=1):
    return (a + b) * c

>>> foo = debug(foo)   # This line becomes unnecessary!
```

This new `@decorator` syntax applies a decorator at the time of function declaration.

Other Uses of Decorators

- Cache function return values (memoization)
- Set function timeouts
- Handle administrative logic (routing, permissions, etc.)

Review – Phew!

- **Functional programming** is a programming paradigm built on the idea of composing programs of functions, rather than sequential steps of execution.
- **`map` and `filter`** simplify common programming patterns.
- **`lambdas`** are smaller, cuter functions, useful to customize the operation of Python functions.
- **Iterators/Generators** are useful for working with infinite – or finite, expensive – streams of data.
- **Decorators** are the neatest thing in the entire world.