UC Berkeley EECS
Lecturer
Michael Ball

Programming Paradigms

Programming Paradigms

- **Paradigm** (Merriam Webster): a typical example or pattern of something; a model. Example: "there is a new paradigm for public art in this country"
- Programming Paradigm (Joe Turner, Clemson University): "A programming paradigm is a general approach, orientation, or philosophy of programming that can be used when implementing a program." *You might call this a "style"*
- Example, three very different approaches to squaring list:

```
map(lambda x: x*x, range(5))
[ x * x for x in range(5) ]
range(5).square_nums() # Only theoretically
```

Why?

- Understanding the paradigm helps you understand the intent of the programmer
- Pick the right tool for the job!
- Most programs written today are multi-paradigm
  - They mix and match the style

Word of Warning

- There is no universally agreed upon taxonomy of human programming styles.

One possible list:

- Imperative
- Functional
- Array-based
- Object oriented
- Declarative

These terms are a bit fluid, and as you'll see if you read more on wikipedia, there is substantial disagreement about these terms.

Programming Paradigms

- Example, three very different approaches to squaring list:
- Functional: map(lambda x: x*x, [1, 2, 3])
- Array-based: [1, 2, 3] * [1, 2, 3] → [1, 4, 9] # Not Python!
- Imperative:

```
def square(nums):
    result = []
    for num in nums:
        result.append(num * num)
    return result
```

The Imperative Programming Paradigm

- An imperative program provides a sequence of steps.
- Like following a recipe.
- Assignment is allowed (can set variables).
- Mutation is allowed (can change variables).
- Example (acronym):

```
def acronym_i(words):
    result = ''
    words = words.split(' ')
    for word in words:
        if len(word) > 4:
            result += word[0]
    return result
```

The Functional Programming Paradigm

- In functional programming, computation is thought of in terms of the evaluation of functions.
- No state (e.g. variable assignments).
- No mutation (e.g. changing variable values).
- No side effects when functions execute.

```
def acronym_f(words):
    return reduce(add,
                  map(lambda w: w[0],
                      filter(lambda w: len(w) > 3,
                             words.split(' '))))
```

Imperative vs. Functional

- Can argue that functional is a subset of imperative.
- Functional programming is still a series of steps.
- Just need to avoid state and think of computation as functions.

- Programs in the functional paradigm:
- More often only one obvious way to do something.
  - Programming feels more like solving puzzles.
  - Solutions can seem like magic (especially to imperative programmers).
- Tend to be shorter.
- Tend to be easier to debug (no need to track variables / side effects).
- Tend to parallelize better (can split work on multiple computers).
  - Example: Each computer can do 1/8th of a “map” operation.
- Are growing in popularity.

A Hybrid Approach

- These paradigms are not official rules. Just attempts to taxonomize approaches taken by humans.
- Code below is sorta functional, sorta imperative.
- Utilizes state for clarity. Many program this way. You might not.

```
def acronym_h(words):
    words = words.split(' ')
    long = filter(lambda w: len(w) > 4, words)
    letters = maps(lambda w: w[0], long)
    return ''.join(letters)
```

Discussion and Debate

- Which of these do you like best?

Less to keep track of.

```
def acronym_f(words):
   return reduce(add,
          map(lambda w: w[0],
          filter(lambda w: len(w) > 3,
                     words.split(' '))))
```

```
def acronym_i(words):
    result = ''
    words = words.split(' ')
    for word in words:
        if len(word) > 4:
             result += word[0]
    return result
```

Very small steps to reason about. Seems "natural", but lots of code

Easy to come back and read it later.

```
def acronym_h(words):
    words = words.split(' ')
   long = filter(lambda w: len(w) > 4,
words)
    letters = maps(lambda w: w[0], long)
    return ''.join(letters)
```

Array-Based Programming!

- Unfortunately not something we can easily demo in Python.
- Treats arrays a "first class" objects – not just containers:
  - Mathematical Operations correspond to "Pairwise" computations: (These are not Python!)

```
[1, 2, 3] * [1, 2, 3] == [1, 4, 9]
[1, 2, 3] + [1, 2, 3] == [2, 4, 6]
```

- Very common in data science, engineering!
  - R (STAT 134), MATALAB, Julia, APL

The Object Based Programming Paradigm

- In object programming, we organize our thinking around objects, each containing its own data, and each with its own procedures that can be invoked.

- We've had plenty of practice here!
- OOP provides many tools!
- But also leaves many import questions open:
  - Should functions be mutalbe or immutable?

Object Based Programming

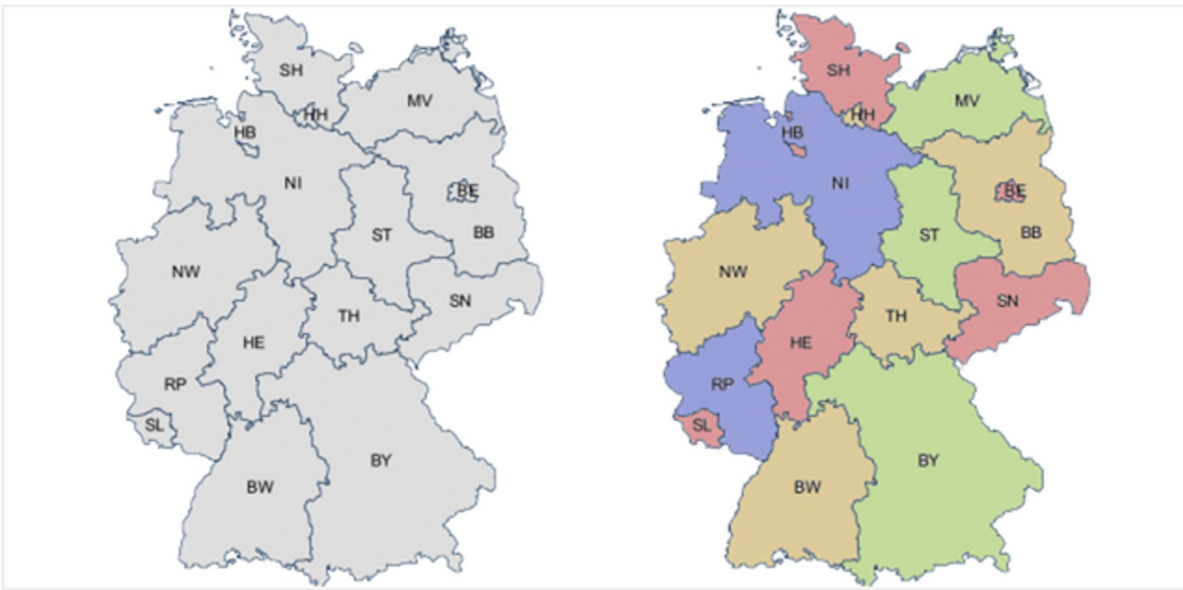
- There is a LOT more than what we see in CS88
  - Rich model for composing classes together
  - Can *easily* be overused.
- In Python "everything is an object"
  - Global functions like len() correspond to "magic" methods on objects, e.g. `__len__`

Declarative Programming

- In declarative programming, we express what we want, without specifying how. A program is simply a description of the result we want.

Declarative Programming

- In declarative programming, we express what we want, without specifying how. A program is simply a description of the result we want.

- Example

anguage:

Prolog Example (From Bernardo Pires)

- Tell Prolog that colors exist:

```
color(red).
color(green).
color(blue).
color(yellow).
```

Tell Prolog that same colors can’t touch:

```
neighbor(StateAColor, StateBColor) :- color(StateAColor),
    color(StateBColor),
    StateAColor \= StateBColor. /* \= is the not equal operator */
```

Tell Prolog all the borders:

```
germany(SH, MV, HH, HB, NI, ST, BE, BB, SN, NW, HE, TH, RP, SL, BW, BY) :-
neighbor(SH, NI), neighbor(SH, HH), neighbor(SH, MV),
neighbor(HH, NI),
neighbor(MV, NI), neighbor(MV, BB),
neighbor(NI, HB), neighbor(NI, BB), neighbor(NI, ST), neighbor(NI, TH),
neighbor(NI, HE), neighbor(NI, NW),
neighbor(ST, BB), neighbor(ST, SN), neighbor(ST, TH),
neighbor(BB, BE), neighbor(BB, SN),
neighbor(NW, HE), neighbor(NW, RP),
neighbor(SN, TH), neighbor(SN, BY),
neighbor(RP, SL), neighbor(RP, HE), neighbor(RP, BW),
neighbor(HE, BW), neighbor(HE, TH), neighbor(HE, BY),
neighbor(TH, BY),
neighbor(BW, BY).
```

Ask Prolog for answer:

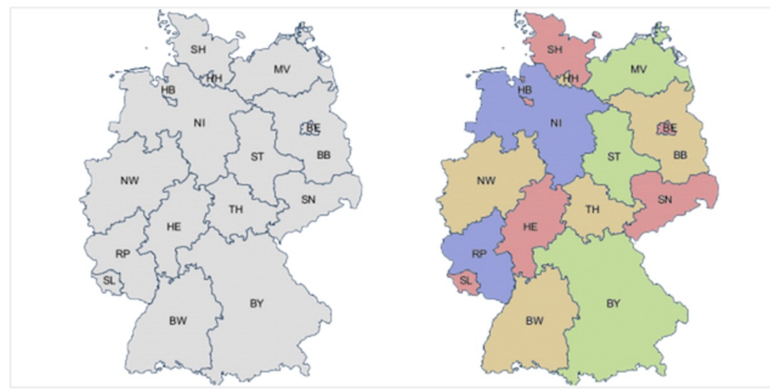
```
?- germany(SH, MV, HH, HB, NI, ST, BE, BB, SN, NW, HE, TH, RP, SL, BW, BY).
```

Declarative Programming

- In declarative programming, we express what we want, without specifying how. A program is simply a description of the **result** we want.

- Another example, coloring a map of Germany using the Prolog language:

BB = BW, **BW** = HB, **HB** = NW, **NW** = SH, **SH** = SL, **SL** = TH, **TH** = red,
BE = NI, **NI** = RP, **RP** = SN, **SN** = green,
BY = yellow,
HE = HH, **HH** = MV, **MV** = ST, **ST** = blue



Declarative Programming

- Each declarative language has only a limited number of tasks for which you can specify "what", and not "how", e.g.
- Prolog: Logic.
- SQL: Queries from a database.
- Pandas: Data manipulation operations like aggregation, filtering, joining, etc.
  - Very common operations in Data 8 and Data 100.
  - Pandas is a library for Python. You'll use it in Data 100!

Why SQL?

- SQL is a ***declarative* programming language** for accessing and modifying data in a relational database.
- It is an entirely new way of thinking ("new" in 1970, and new to you now!) that specifies *what* should happen, but not *how* it should happen.
- Python is a multi-paradigm language, but we haven't yet tried declarative programming.

What is SQL?

- A declarative language
  - Described *what* to compute
  - Query processor (interpreter) chooses which of many equivalent query plans to execute to perform the SQL statements
- ANSI and ISO standard, but many variants
  - CS88's SQL will work on nearly all relational databases—databases that use tables. [We'll revisit next lecture!]
- `SELECT` statement creates a new table, either from scratch or by projecting a table
- `CREATE TABLE` statement gives a global name to a table
- Lots of other statements
  - `analyze, delete, explain, insert, replace, update, …`

SQL: Describe The Shape of the result!

```
# An example of creating a Table from a list of rows.
Table(["Flavor","Color","Price"]).with_rows([
    ('strawberry','pink', 3.55),
    ('chocolate','light brown', 4.75),
    ('chocolate','dark brown', 5.25),
    ('strawberry','pink',5.25),
    ('bubblegum','pink',4.75)])
```

Flavor	Color	Price
strawberry	pink	3.55
chocolate	light brown	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
bubblegum	pink	4.75

What if I want a table with just a few rows?

- Here the `where()` in Python is using the datascience module.

```
sqlite> select * from cones where Flavor = "chocolate";
ID|Flavor|Color|Price
2|chocolate|light brown|4.75
3|chocolate|dark brown|5.25
6|chocolate|dark brown|5.25
```

```
cones.where(cones["Price"] > 5)
```

ID	Flavor	Color	Price
3	chocolate	dark brown	5.25
4	strawberry	pink	5.25
6	chocolate	dark brown	5.25

SQL:

```
sqlite> select * from cones where Price > 5;
ID|Flavor|Color|Price
3|chocolate|dark brown|5.25
4|strawberry|pink|5.25
6|chocolate|dark brown|5.25
```

Summary

- Paradigms are styles, guidelines for how to approach a program
- Each is equally capable, but some are suited best to particular tasks.
- Declarative programming gets us to think about the *what* rather than the *how*.