Computational Structures in Data Science

Programming Paradigms

Berkeley
UNIVERSITY OF CALIFORNIA

CITN: A "Backdoor" Almost Infected Millions of Computers

What we know about the xz Utils backdoor that almost infected the world [Link]

Malicious updates made to a ubiquitous tool were a few weeks away from going mainstream.

DAN GOODIN - 3/31/2024, 11:55 PM

"On Friday, a lone Microsoft developer rocked the world when he revealed a backdoor had been intentionally planted in xz Utils, an open source data compression utility available on almost all installations of Linux and other Unix-like operating systems. The person or people behind this project likely spent years on it. They were likely very close to seeing the backdoor update merged into Debian and Red Hat, the two biggest distributions of Linux, when an eagle-eyed software developer spotted something fishy."

Why is this interesting? Software is incredibly complex, managed by many individuals. Open-source software (like Python, Jupyter, Linux) means everyone can read + audit the source code

Announcements

- SQL – Next 3 Lectures
- Ants Check Offs
  - Requires you to have written the code but trying not to be stressful.
  - Don't use LLMs for the Reflection questions.
- Review/Wrap Up + Q&A April 29
- Don't forget to do your taxes! :/
  - But you'll probably get a refund, and can even file for free
    - We don't have time for it – but good story of tech + government + society

Computational Structures in Data Science

Back to Iterators

Berkeley
UNIVERSITY OF CALIFORNIA

What's an Iterator? [Docs]

iterator

An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead.

iterable

An object capable of returning its members one at a time. Examples of include all sequence types and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements sequence semantics.

Next element in generator iterable

- Iterables work because they implement some "magic methods" on them. We saw magic methods when we learned about classes,
  - e.g., __init__, __repr__ and __str__.
- The first one we see for iterables is `__next__`

- `iter()` – transforms a sequence into an iterator
  - Usually this is not necessary, but can be useful.

Iterators: The `iter` protocol [Docs]

- In order to be iterable, a class must implement the iter protocol
- The iterator objects themselves are required to support the following two methods, which together form the iterator protocol:
  - __iter__: Return the iterator object itself. This is required to allow both containers and iterators to be used with the for and in statements.
    - This method returns an iterator object (which can be self)
  - __next__ : Return the next item from the container. If there are no further items, raise the StopIteration exception.

The Iter Protocol In Practice

- Classes get to define how they are iterated over by defining these methods
  - containers (objects like lists, tuples, etc) typically define a Container class and a separate ContainterIterator class.
- Lists, Ranges, etc are *not* directly iterators
  - We cannot call next() on them.
  - However, they implement an `__iter__` method, and `list_iterator`, `range_iterator` class, etc.

Iterables

Demo

Computational Structures in Data Science

Building a Range Iterator

Berkeley
UNIVERSITY OF CALIFORNIA

Making a Range Iterator

- What does a range need?
  - Start value
  - Stop
  - (We'll ignore step sizes)
- keep track of the current value
- An `__iter__` method
- A `__next__` method

Example

```
class myrange:
    def __init__(self, n):
        self.i = 0
        self.n = n
    def __iter__(self):
        return self
    def __next__(self):
        if self.i < self.n:
                ----------------------------------------
                ----------------------------------------
                ----------------------------------------
          else:
                raise StopIteration()
```

Example

```
class myrange:
    def __init__(self, n):
        self.i = 0
        self.n = n
    def __iter__(self):
        return self
    def __next__(self):
        if self.i < self.n:
            current = self.i
            self.i += 1
            return current
        else:
                raise StopIteration()
```

Bonus! CountIterator

```
class CountIterator:
    def __init__(self, iterator):
        self._it = iter(iterator)
        self._count = 0
    def __iter__(self):
        return self
    def __next__(self):
       …
    def num_calls(self):
        """Return the number of items yielded so far."""
        return self._count
```

Computational Structures in Data Science

The GetItem Protocol

Berkeley
UNIVERSITY OF CALIFORNIA

(Optional) Get Item protocol

- Another way an object can behave like a sequence is indexing: Using square brackets "[ ]" to access specific items in an object.
- Defined by special method: `__getitem__(self, i)`
  - Method returns the item at a given index

```
class myrange2:
    def __init__(self, n):
        self.n = n

    def __getitem__(self, i):
        if i >= 0 and i < self.n:
            return i
        else:
            raise IndexError

    def __len__(self):
        return self.n
```

Computational Structures in Data Science

Iterators and Generators Review

Berkeley
UNIVERSITY OF CALIFORNIA

Michael Ball | UC Berkeley | https://c88c.org |

Terms and Tools

- **Iterators:** Objects which we can use in a for loop
  - Anything that can be looped over!
  - Sometimes they're lazy, sometimes not!
- **Generators:** A shorthand way to make an iterator that uses yield
  - a function that uses `yield` is a *generator function*
  - a generator function returns a *generator object*
  - Generators do **not** use return
- **Sequences:** A particular type of iterable
  - They know they're length, support slicing
  - Are *not* lazy

Computational Structures in Data Science

(Optional) Type Checking

Berkeley
UNIVERSITY OF CALIFORNIA

(Optional) Determining if an object is iterable

```
from collections.abc import Iterable
isinstance([1,2,3], Iterable)
from collections.abc import Sequence
isinstance([], Sequence)        # True
isinstance("hi", Sequence)      # True
isinstance(range(5), Sequence) # True
isinstance({1, 2}, Sequence)    # False
isinstance({}, Sequence)        # False
isinstance(iter([]), Sequence)  # False
```

- This is more generic than checking for any items of particular type, e.g., list, tuple, string…

Computational Structures in Data Science

Programming Paradigms

Berkeley
UNIVERSITY OF CALIFORNIA

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"

Why?

- Understanding the paradigm helps you understand the intent of the programmer
- Pick the right tool for the job!
  - Different problems require different solutions
- Most programs written today are multi-paradigm
  - They mix and match the style
- Problem solving technique

Examples of Paradigms

Example, three very different approaches to squaring list:

```
lst = []
for i in range(5):
        lst += [ i*i ]

map(lambda x: x*x, range(5))

[ x * x for x in range(5) ]
range(5).square_nums() # Only theoretically, e.g assume `def square_nums(self)` exists.
```

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:

```
np.array([1,2,3]) * np.array([1,2,3])
np.array([1,2,3]) ** 2
```

Imperative:

```
def squares(nums):
    result = []
    for num in nums:
        result += [ 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.
- Functional Programs:
- More often fewer clear /correct ways to do something.
  - Programming feels more like solving puzzles.
  - Solutions can seem like magic (especially to imperative programmers).

Why do we push functional programming?

- 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.
  - Reducing mutations makes computation easier to scale
  - Hugely prevalent in AI fields.
- Growing in popularity.
  - Explosion of ideas in new programming languages
  - "old" ideas are becoming new/popular

A Hybrid Approach

- 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?

(A) Imperative
Very small steps to reason about.
Seems "natural", but lots of code

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

(B_ Functional: Less to keep track of. Fewer variables, lines

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

(C)Hybrid: Some functional, some OOP, uses variables

```
acronym_h(words):
    words = words.split(' ')
    long = filter(lambda w: len(w) > 3, words)
    letters = maps(lambda w: w[0], long)
    return ''.join(letters)
```


Array-Based Programming!

- Not something we can easily demo in *native* Python.
- Treats arrays a "first class" objects – not just containers:
- Mathematical Operations correspond to "Pairwise" computations:
  - `np.array([1,2,3]) * np.array([1,2,3])`
  - `np.array([1,2,3]) + np.array([1,2,3]) == [2, 4, 6]` → `array([ True,  True,  True])`
  - **Note!** Even == is now an array operation. Good? Bad? Just different!
- Very common in data science, engineering!
  - R (STAT 20, STAT 134), MATALAB, Julia, APL

Computational Structures in Data Science

Object-Oriented Programming

Berkeley
UNIVERSITY OF CALIFORNIA

The Object-Oriented 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 mutable or immutable?
  - How much inheritance is the right amount?

Object-Oriented Programming

- There is a LOT more than what we see in C88C
  - Rich model for composing classes together
    - Python allows you to inherit from multiple classes at once
  - Can **easily** be overused.
  - Explored in depth in CS61B
- In Python "everything is an object"
  - You benefit from OOP ideas even when you don't realize.
  - Global functions like len() delegate to "magic" methods on objects, e.g. `__len__`

Computational Structures in Data Science

Declarative Programming

Berkeley
UNIVERSITY OF CALIFORNIA

Declarative Programming

- In declarative programming, we express what we want, without specifying how. A program is simply a description of the result we want.
- Can be a very different thought process!
- Incredibly useful, but not necessarily best for all types of problems.

The Web: HTML

- Web pages are built with a language called HTML.
  - Programmers specific what content should be on the page, and where.
  - The browser lays out the content on each device in the right spot for each screen size, etc.
    - Developers don't have to specify what happens when someone changes the window size, or hits print, etc.
  - Tags, like "section", "p" (paragraph), "header" "time" describe the *type of content*

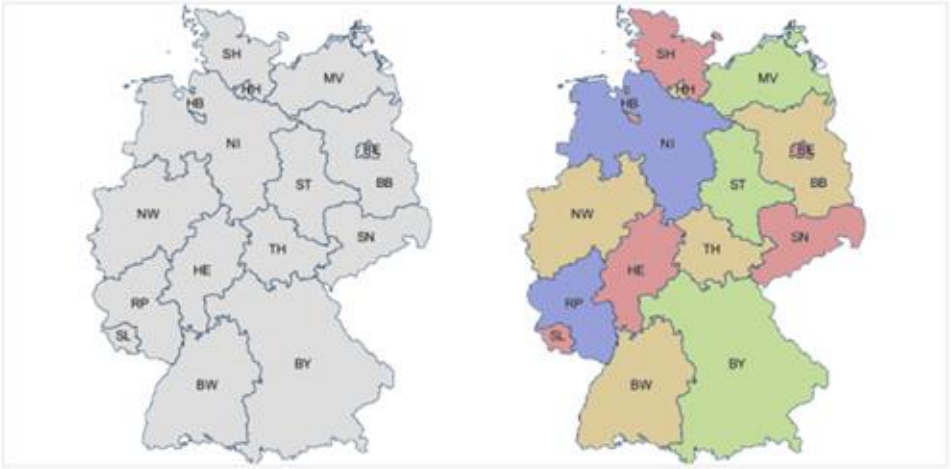
HTML Continued

- A partial section of the CS88 Website:

```
<div id="content" class="container">
    <div class="page-header">
    <h1><span class="content-title-brand">CS
88</span>:
         Computational Structures in Data Science
    <div class="small">Fall 2023</div>
    <div class="small">Instructor: Michael Ball</div>
   </h1>
</div>
<section><h2>Announcements</h2>…
```

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: coloring a map of Germany using the Prolog language:



Prolog Example (From Bernardo Pires)

- Tell Prolog that colors exist:
  Tell Prolog that same colors can't touc

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

```
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 → Results

- Result is a list of states and color pairs

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 and `datascience` modules: Data manipulation operations like aggregation, filtering, joining, etc.
  - Very common operations in Data 8 and Data 100.
  - While the syntax of Pandas is odd, the ideas will build upon Data 8.

Declarative Programming In Data 8

- `cones.group('Flavor')`
  - `datascience` module figures out the grouping
- `table.where(label, conditions)`
- Can combine these simpler expressions together for more complex questions

Declarative or Object-Oriented?

- Both!
- Tables (in Data 8, Pandas, etc) are Python objects
  - There is a `class Table` with a `def columns(self)` method
- However, the *interface* is *often* declarative.
  - You describe what the output should look like

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.

What is SQL?

- SELECT statement creates a new table, either from scratch or by projecting a table
- INSERT adds to a table, UPDATE changes data.
- CREATE TABLE statement gives a global name to a table
- Lots of other statements
  - ANALYZE, DELETE, EXPLAIN, ...

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.

```
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
```

```
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
```


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.
- Almost no programs are purely single-paradigm

