Python for Interviews, Automation, & Data Science

JD Kilgallin

CPSC:480

11/23/22

Pressman Appendix B

Learning Objectives

- Fundamental concepts of Python development
- Applications of Python for software engineers

- *Slides 12, 18, and 22 (marked "Examinable" in red) are examinable*

Python

- Python is a general-purpose, high-level, interpreted language, meaning it's not compiled; each statement is evaluated in order by the Python interpreter.
- It is named for Monty Python, not the type of snake. I did have a pet python named Java, though.
- It supports object-oriented programming, functional programming, and other paradigms, with a large standard library & set of packages.
- It is *dynamically typed* – variables don't have a type until runtime.
- It is a powerful language that emphasizes expressiveness, readability, and community development.
- There are major differences between Python 2.x & Python 3.x syntax

Running Python

- You can install command-line tools from python.org (or from e.g. any Linux package manager), or configure support in your IDE of choice.
- Running "python" will start an interactive shell, where you can start typing python code and see the results right away. This makes a great general-purpose calculator.
- Most Python code will be entered in a Python script file (typically a ".py" extension) and "python myScript.py" will execute the script.
- Python files can be referenced from other files and optionally packaged into modules. "import <file>" works similarly to #include.

Python syntax

- Indentation levels are used instead of {} to denote control-flow blocks
- "if" and "for" statements end with a colon. "and"/"or"/"not" get written out, and constants "True", "False", and "None" for Booleans and nulls.
- Tons of built-in operators. For example, exponent "**": 2 ** 5 == 32. String multiplication: "abc" * 3 == "abcabcabc". Base conversion: int("a",16)==10.
- You can assign/reassign variables to different types without declaring type.
- "import foo" imports a local .py file or standard library module "foo"

```
x = "Hello, world"
print(x)
x = 2+2
if x > 3 and x < 5:
    print("x is now 4")
 print("Done with that 'if' block")
# prints all three strings on separate lines. Oh yeah, '#' starts a comment.
```

Data structures

- Type system is very similar to JSON – primitives like int, bool, and string can be structured in arrays, tuples (fixed-length arrays), and dictionaries. These use square brackets [], parentheses (), and curly braces {}, respectively.

```
x = [0, 2.5, False, "Charizard"]
print(x[3])  # Prints "Charizard"
x = {"jdk72":"JD Kilgallin", "aa1":"Aaron Aardvark"}
print(x["jdk72"])  # Prints "JD Kilgallin"
```

Slices and iteration

- Can take a range of an array or string and return the sub-array/string within those indexes. Can use negative indexes to access the end of the array/string. Can optionally use a third "step" parameter.
  - print("myScript.py"[:-3]) gives "myScript"
  - If myString[-3:] == ".py" # Same as myString.endsWith(".py")
  - print("Hello"[::-1]) # prints "olleH"
- "len(x)" function gives length of an array or string.
- Can use "for x in myArray:" syntax to iterate through an array or string.
  - Use "range(n)" to get a list of the first n integers (starting at 0)

```
x = [2,4,8,16]
for i in range(10):
    if i not in x:
        print(i)  # prints 0, 1, 3, 5, 6, 7, 9
```

Comprehensions

- Data structures should be thought of as first-class data types, and code should operate on the structure as a whole rather than iterating through and operating element-by-element.
- The above is true in every language, but Python allows the easy construction of an array or dictionary out of the results of a loop, called a "comprehension", that forces this way of thinking effectively. This provides a way to write really good code that's very short.

```
x = [i ** 2 for i in range(6)]  # [0,1,4,9,16,25]
jd = {"name":"JD Kilgallin", id":"jdk72","email":"jdk72@uakron.edu"}
joe = {"name":"Joe Smith","id":"jis26","email":"jis26@uakron.edu"}
jim = {"name":"Jim Wheeler","id":"jw4","email":"jw4@uakron.edu"}
students = {s.id:s for s in [jd,joe,jim]}  # Key students by ID
students["jdk72"].name  # Prints "JD Kilgallin"
```

Functions

- Functions are defined with "def functionName(param1, param2):"
- Arguments can be passed positionally (pass them in the same order they're defined) or by name – myFunction(param2="foo",param1= "bar"). Default values can be assigned in function definition to make parameters optional.
- Functions can be passed as parameters by name.
- Lambda expressions can be used to define anonymous functions.

```
def apply(function,param):
    return function(param)

apply(lambda x: 2 ** x, 4)  # Returns 2^4=16
```

Functional programming

- "map(function, list)" applies "function" to every element of "list".
  - map(lambda x: x ** 2, range(6)) # gives 1,4,9,16,25,36
- "filter(predicate, list)" gives a representation of the elements of "list" where "predicate" returns True for that element.
  - filter(lambda x: x % 2 == 1, range(6)) # gives 1,3,5
  - Can be combined with "next" to return the first element matching predicate. next(filter(lambda x: x ** 2 > 10, range(5))) # returns 4
- "groupby(list, selectorFunction)" returns a dictionary where the keys are the unique results of applying the selector function to elements of the list, and the value for each key contains all the elements in the list where the selector returns that key. List must be sorted though.
  - # Sort 0..5 by value mod 3, then group them that way. {0:[0,3],1:[1,4],2:[2]} groupby(sorted(range(5),key=lambda x:x%3), lambda x: x % 3)

Reduce

- "reduce(accumulator, list, initial)" applies an accumulator function that takes two arguments and returns a value that goes into the next call.
- The first argument is the cumulative result of previous calls to the accumulator, and the second is the next element in the list. In the first call, "initial" is used as the first value.
  - e.g. reduce(lambda x,y:x+y, [4,10,60], 0) calculates 0+4=4, then 4+10=14, then 14+60=74, summing the whole list.
- The final output is the return value of the last accumulator call.

```
def redact(text, myString):  # Removes instances of myString from text
    return text.replace(myString,"REDACTED")
secrets = ["aliens","9/11"]  # Terms to censor in output
reduce(redact, secrets, "We have proof that aliens committed 9/11")
# Prints "We have proof that REDACTED committed REDACTED"
```

Example – REST APIs

Examinable

- Many applications define an interface that can be accessed over HTTP web requests.
- REpresentational State Transfer (REST) is a paradigm for standardizing request and response formats to access resources via web API.
- URLs are grouped based on resource types, and use nouns to identify resources. HTTP verbs "POST", "GET", "PUT", and "DELETE" define Create/Read/Update/Delete operations, respectively.
  GET https://keyfactor.local/Users/1/Roles could be a REST endpoint for retrieving the roles for user with id "1".
- The OpenAPI Initiative defines JSON and YAML formats for representing a RESTful API, called its "OpenAPI specification" (formerly "Swagger"). An OpenAPI specification defines the API with enough detail that it can be consumed to programmatically access the API – client code can be generated in any language to make HTTP requests with correct input/output data.

Example – Partial Keyfactor OpenAPI

```
"paths": {                          Defines the set of URLs. Each URL is a key, and the value is info about the endpoint.
  "/Agents/{id}": {
    "get": {                        Definition for e.g. GET /Agents/1
      "tags": [
        "Agent"
      ],
      "summary": "Returns details for a single agent, specified by ID",
      "operationId": "Agent_GetAgentDetail",          Identifier for this endpoint. Must be unique.
      "consumes": [],               Request body content types
      "produces": [                 Response body content types
        "application/json",
        "application/xml"
      ],
      "parameters": [               Parameters sent in requests to this endpoint
        {
          "name": "id",
          "in": "path",             "id" parameter is in the URL, not a query parameter or passed in request body.
          "description": "Agent Id to Search",
```

Example Python – Make operationIDs Unique

```
# HTTP method types for calls to Keyfactor REST API
methods = ["get","post","put","delete"]
# JSON translates directly to Python dictionary with json.loads()
paths = json.loads(openAPISpecificationContents)["paths"]
# Loop through all the URL paths. For each path, take the set of methods the URL supports. Map the operation ID to the URL-method combo.
opGroups = [{paths[p][m]["operationId"]:{p:m}}for p in paths.keys() for m in filter(lambda x: x in paths[p],methods)]
# Group URL-method combos by operation ID and map each operation id to the list of methods with that operation id
opMap = {k:[x[k] for x in list(g)] for k, g in itertools.groupby(sorted(opGroups, key=firstKey), firstKey)}
# Any operation ID that maps to more than one URL-method combo indicates a duplicate operation ID that must be corrected so they're unique.
duplicates = [v for (k,v) in opMap.items() if len(v) > 1]
```

Strengths of Python

- Expressive syntax allows for very short code.
- Data structure and function manipulation is unparalleled.
- Dynamic type system allows for easy function overloading, variable reuse, and heterogeneous data structures.
- Syntactically significant whitespace eliminates lines that contain a single brace.
- Interpreter means code is portable across platforms.
- Interpreter means code can easily be manually tested live. You can call your functions with desired arguments from the interactive shell.

Python for Interviews

- The strengths of Python make it excellent for short, leet-code style data structure problems.

```
# twoSum in O(n) - arr.count(), comprehensions, and filter are all O(n)
def twoSum(target,arr):
    if arr.count(target/2) > 1: # Get the first and last index of target/2
        return (arr.index(target/2),len(arr)-1-arr[::-1].index(target/2))
    # Map each element in input array to its (last) index in the array
    index = {arr[i]:i for i in range(len(arr))}
    # Compute ALL pairs that add to target, excluding same-index pairs
    pairs = [(index[i],index[target-i]) for i in index if target-i in index]
    return next(filter(lambda x: x[0] != x[1],pairs),None)
```

Python for Automation

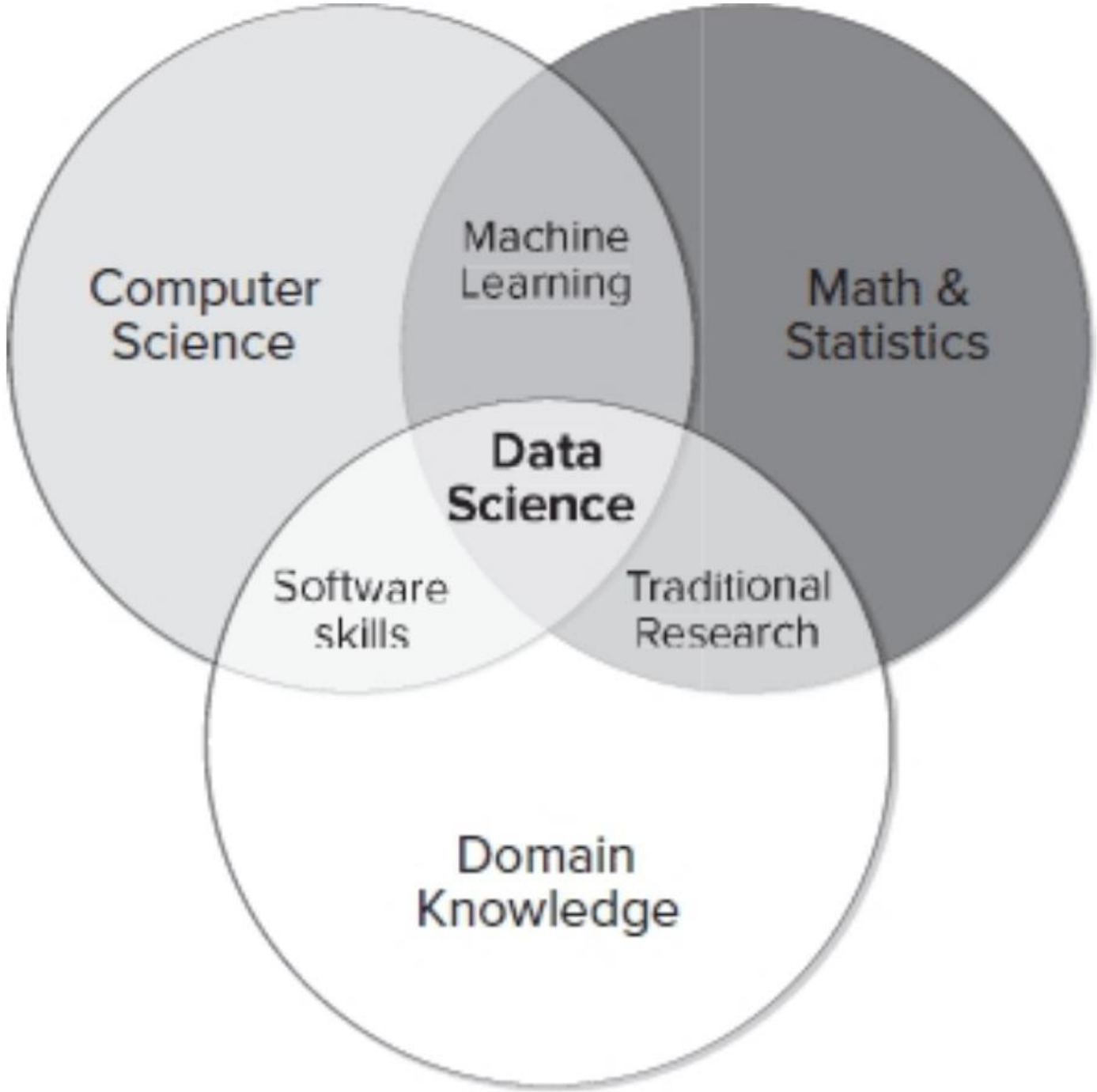
- "system()" function and "os" module allow easy execution of command-line instructions with output that can be easily parsed.
- Once again, Python's expressiveness and standard library makes it very well suited for short scripts with arbitrary capabilities.

```
# Practice final asked for a method to find matching filenames with lowest possible cyclomatic complexity. With Python data structures, this can be done with CC=1!
import os
def getContents(path):
    if os.path.isfile(path): return []
    return os.listdir(path)
def find(target, path="/"):
    allPaths = getContents(path) + reduce(lambda x,y:x+y,map(lambda x:find(target,path+"\\"+x), getContents(path)),[])  # Concatenate paths recursively
    return list(filter(lambda x: target in x,allPaths))  # Return paths with a match
```

Data Science

Examinable

- Data science is, broadly, the transformation of raw data into useful, actionable information.
- Data science requires the simultaneous application of computer science, math/statistics, and domain knowledge to identify patterns.
- Data science is valuable for businesses to gain actionable insights from data they've collected. For example, Netflix can use data science to recommend shows based on what else you've liked, and what other shows are correlated with the ones you've liked based on other users' preferences. This helps Netflix retain customers.
- The problems and techniques in data science can be very broad, but they help to provide insights to problems that don't have an exact algorithmic solution (like predicting Netflix preferences).


Computer Science
Machine Learning
Math & Statistics
Data Science
Software skills
Traditional Research
Domain Knowledge

Python for Data Science

- Python is exceptionally good at transforming data sets in disparate formats and combining them into a cohesive aggregated data set.
- The fact that the Python shell can be run interactively makes it easy to explore aggregated data sets in a very flexible manner.
- Data analysis tasks tend to be short programs that involve a lot of data structure manipulation, which is where Python shines.
- The Python PANDAS library, in particular, is one of the most popular data science libraries in the world. Many other libraries such as NumPy and Matplotlib assist with data science tasks too.

Weaknesses of Python

- The Python interpreter is slow, and Python programs don't run as fast as native code written in languages like C/C++.
- Dense code can be hard to read in some cases, even if it is expressive.
- Using indentation for scope can easily trip up programmers that aren't accustomed to it, leading to errors that can be hard to find.
- Dynamic typing means that many logic errors can't be caught at compile time, and may not always even be caught at run-time, leading to unexpected results that can be hard to verify.
- Interpreted nature and dynamic typing means that impact of a change on other code can be hard to assess in a large project.
- Maintenance of large programs is very difficult, and thus Python is not well-suited for large software engineering projects.

Summary

Examinable

- Python is an expressive, interpreted language that is well-suited for short programs, especially ones involving data structure manipulation
- The functional programming constructs such as lambda functions, comprehensions, and map/filter/reduce/groupby functions reinforce a very clean coding style that engineers should follow in many other cases with other languages.
- Python is much less suitable for complex software engineering projects as the slow interpreter and dynamic typing make it difficult to assess impacts of code changes.
- Major applications of Python include software engineering interview questions, automation scripts, and data science analytics.

References

- Python. Python Software Foundation. 2001-2022.
- Python (programming language). Wikipedia.
- Python for Programmers. JD Kilgallin. 2016. Keyfactor.