Unit 5: An Introduction to Testing

Unit 6: Using Linters to Support Python Testing

NOTES

Unit 5: An Introduction to Testing

Pillai (2018), chapter 3

The testability can be defined as follows: "The degree of ease with which a software system exposes its faults through execution-based testing“

The different aspects that usually fall under software testing: •

Functional testing: This involves testing the software for verifying its functionality. A unit of software passes its functional test if it behaves exactly the way it is supposed to as per its development specifications. Functional testing is usually of two types:

° White-box testing: These are usually tests implemented by the developers, who have visibility into the software code, themselves. The units being tested here are the individual functions, methods, classes, or modules that make up the software rather than the end user functionality. The most basic form of white-box testing is Unit testing. Other types are integration testing and system testing.

° Black-box testing: This type of testing is usually performed by someone who is outside the development team. The tests have no visibility into the software code, and treat the entire system like a black box. Black-box testing tests the end user functionality of the system without bothering about its internal details. Nowadays, a lot of black-box tests on web-based applications can be automated by using testing frameworks like Selenium.

Performance testing: Tests which measure how a software performs with respect to its responsiveness and robustness (stability) under high workloads come within this category. Performance tests are usually categorized into the following: ° Load testing: Tests that assess how a system performs under a certain specific load, either in terms of the number of concurrent users, input data, or transactions. ° Stress testing: Tests the robustness and response of the system when some inputs present a sudden or high rate of growth and go to extreme limits. Stress tests typically tend to test the system slightly beyond its prescribed design limits. A variation of stress testing is running the system under a certain specified load for extended periods of time, and measuring its responsiveness and stability

Scalability testing: Measure how much the system can scale out or scale up when the load is increased. For example, if a system is configured to use a cloud service, this can test the horizontal scalability—as in how the system auto scales to a certain number of nodes upon increased load, or vertical scalability—in terms of the degree of utilization of CPU cores and/or RAM of the system.

Cont…

Security testing: Tests that verify the system's security fall into this category. For web-based applications, this usually involves verifying authorization of roles by checking that a given login or role can only perform a specified set of actions and nothing more (or less). Other tests that fall under security would be to verify proper access to data or static files to make sure that all sensitive data of an application is protected by proper authorization via logins.

Usability testing: Usability testing involves testing how much the user interface of a system is easy to use, is intuitive, and understandable by its end users. Usability testing is usually done via target groups comprising selected people who fall into the definition of the intended audience or end users of the system.

Installation testing: For software that is shipped to the customer's location and is installed there, installation testing is important. This tests and verifies that all the steps involved in building and/or installing the software at the customer's end work as expected. If the development hardware differs from the customer's, then the testing also involves verifying the steps and components in the end user's hardware. Apart from a regular software installation, installation testing is also important when delivering software updates, partial upgrades, and so on

Accessibility testing: Accessibility, from a software standpoint, refers to the degree of usability and inclusion of a software system towards end users with disabilities. This is usually done by incorporating support for accessibility tools in the system, and designing the user interface by using accessible design principles.

Check the Web Content Accessibility Guidelines (WCAG) of W3C, Regression testing, Acceptance testing, Alpha or Beta testing, and so on.

Some strategies to improve the predictability of the code under test:

• Correct exception handling: – Missing or improperly-written exception handlers is one of the main reasons for bugs and thence, unpredictable behavior in software systems. It is important to find out places in the code where exceptions can occur, and then handle errors. Most of the time, exceptions occur when a code interacts with an external resource such as performing a database query, fetching a URL, waiting on a shared mutex, and the like.

• Infinite loops and/or blocked wait: When writing loops that depend on specific conditions such as availability of an external resource, or getting handle to or data from a shared resource, say a shared mutex or queue, it is important to make sure that there are always safe exit or break conditions provided in the code. Otherwise, the code can get stuck in infinite loops that never break, or on never-ending blocked waits on resources causing bugs which are hard to troubleshoot and fix.

• Logic that is time dependent: When implementing logic that is dependent on certain times of the day (hours or specific weekdays), make sure that the code works in a predictable fashion. When testing such code, one often needs to isolate such dependencies by using mocks or stubs.

• Concurrency: When writing code that uses concurrent methods such as multiple threads and/or processes, it is important to make sure that the system logic is not dependent on threads or processes starting in any specific order. The system state should be initialized in a clean and repeatable way via well-defined functions or methods which allow the system behavior to be repeatable, and hence, testable.

• Memory Management: A very common reason for software errors and unpredictability is incorrect usage and mismanagement of memory. In modern runtimes with dynamic memory management, such as Python, Java, or Ruby, this is less of a problem. However, memory leaks and unreleased memory leading to bloated software are still very much a reality in modern software systems

Different Testing Libraries in Python

example of patching functions using the Mock library of unittest.

the two other well-known testing frameworks in Python, namely, nose2 and py.test.

examples of measuring code coverage using the coverage.py package directly, and also by using it via plugins of nose2 and pytest.

textsearch class for using advanced mock objects, where we mocked its external dependency and wrote a unit test case.

Python doctest support of embedding tests in the documentation of classes, modules, methods, and functions via the doctest module while looking at examples.

integration tests, and the three different ways in which tests can be integrated in a software organization.

Test automation via Selenium and the TDD principles

In Python, support for unit testing is provided by the unittest module in the standard library.

There are other unit-testing modules in Python which are not part of the standard library, but are available as third-party packages.

The py.test package, commonly known as pytest, is a full-featured, mature testing framework for Python. Like nose2, py.test also supports test discovery by looking for files starting with certain patterns.

Measuring coverage using coverage.py

Pillai, chapter 4 (133)

The performance of a software system can be broadly defined as: "The degree to which the system is able to meet its throughput and/or latency requirements in terms of the number of transactions per second or time taken for a single transaction.“

Software performance engineering includes all the activities of software engineering and analysis applied during the Software Development Life Cycle (SDLC) and directed towards meeting performance requirements

Performance testing and diagnostic tools can be classified further as follows:

Examples of common stress testing tools used for web application testing include httpperf, ApacheBench, LoadRunner, Apache JMeter, and Locust.

Performance testing and diagnostic tools

Monitoring tools: These tools work with the application code to generate performance metrics such as the time and memory taken for functions to execute, the number of function calls made per request-response loop, the average and peak times spent on each function, and so on.

Instrumentation tools: Instrumentation tools trace metrics, such as the time and memory required for each computing step, and also track events, such as exceptions in code, covering such details as the module/function/ line number where the exception occurred, the timestamp of the event, and the environment of the application (environment variables, application configuration parameters, user information, system information, and so on). Often external instrumentation tools are used in modern web application programming systems to capture and analyze such data in detail.

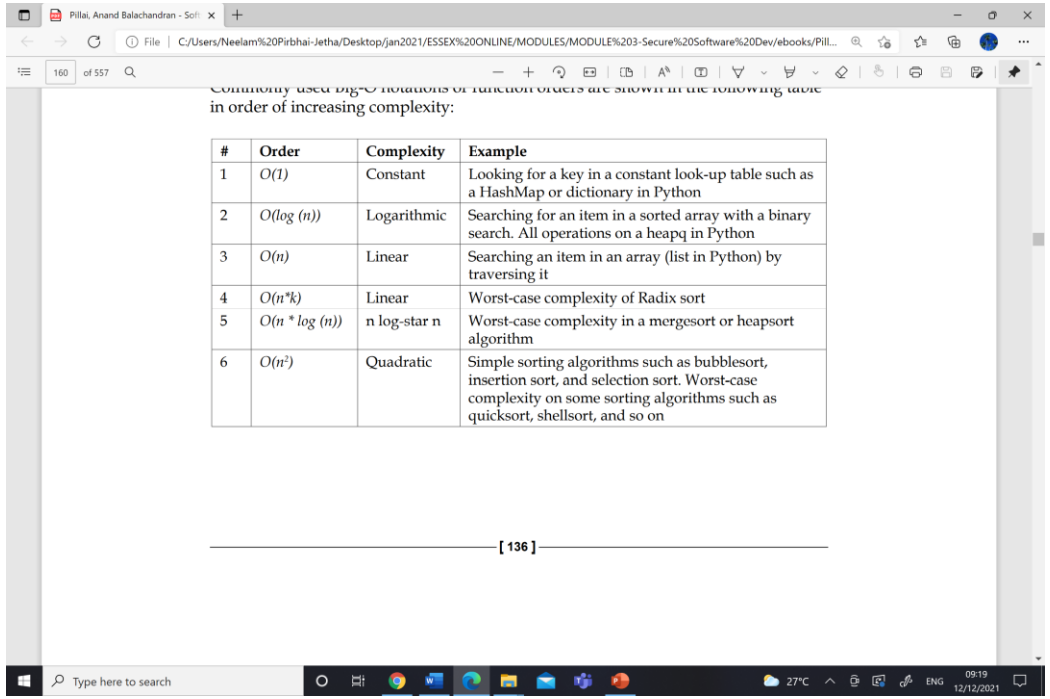
Performance testing and diagnostic tools

Code or application profiling tools: These tools generates statistics about functions, their frequency of duration of calls, and the time spent on each function call. This is a kind of dynamic program analysis. It allows the programmer to find critical sections of code where the most time is spent, allowing him/her to optimize those sections. Optimization without profiling is not advised as the programmer may end up optimizing the wrong code, thereby not surfacing the intended benefits up to the application.

In Python, the available set of tools in the standard library: the profile and cProfile modules

Performance complexity

This is usually represented by the so-called Big-O notation which belongs to a family of notations called the Bachmann–Landau notation or asymptotic notation.

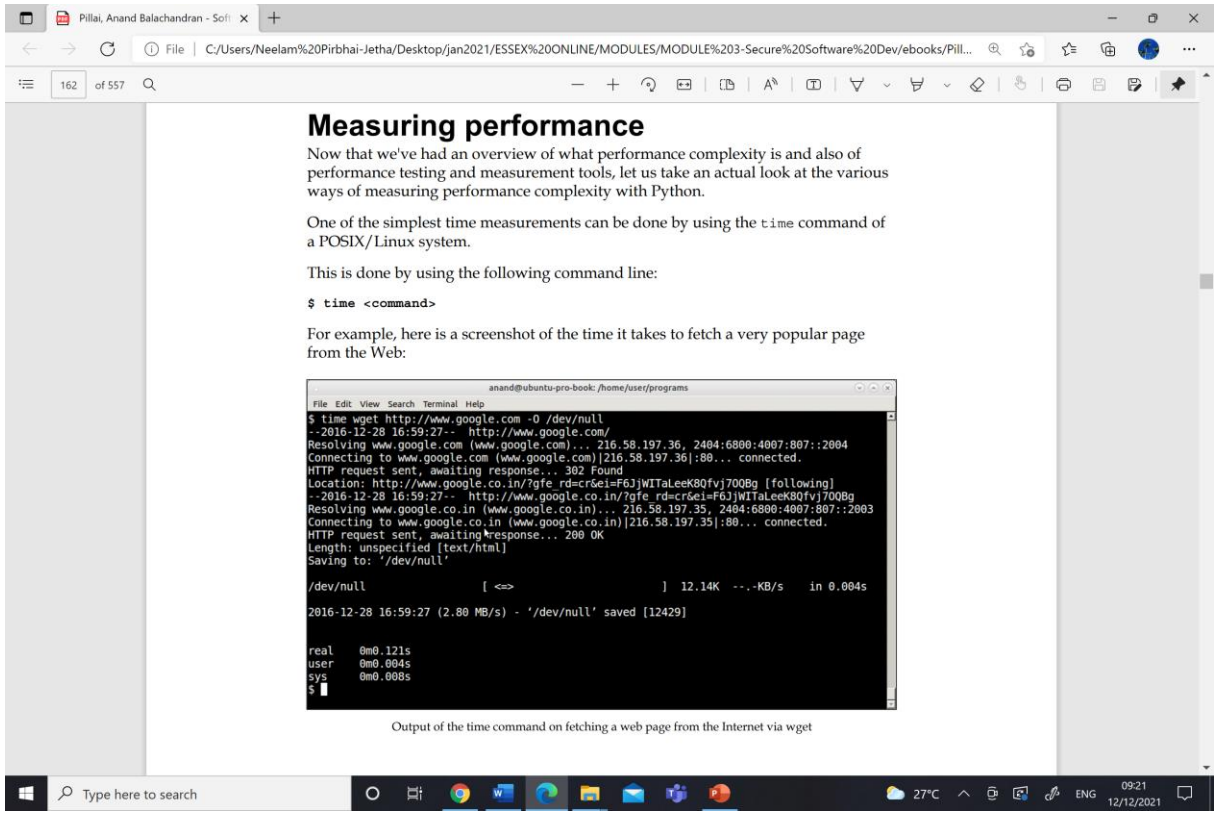


in order of increasing complexity:

#	Order	Complexity	Example
1	*O(1)*	Constant	Looking for a key in a constant look-up table such as a HashMap or dictionary in Python
2	*O(log (n))*	Logarithmic	Searching for an item in a sorted array with a binary search. All operations on a heapq in Python
3	*O(n)*	Linear	Searching an item in an array (list in Python) by traversing it
4	*O(n*k)*	Linear	Worst-case complexity of Radix sort
5	*O(n * log (n))*	n log-star n	Worst-case complexity in a mergesort or heapsort algorithm
6	$O(n^2)$	Quadratic	Simple sorting algorithms such as bubblesort, insertion sort, and selection sort. Worst-case complexity on some sorting algorithms such as quicksort, shellsort, and so on

Various ways of measuring performance complexity with Python.

One of the simplest time measurements can be done by using the time command of a POSIX/Linux system. This is done by using the following command line: $ time



Measuring performance

Now that we've had an overview of what performance complexity is and also of performance testing and measurement tools, let us take an actual look at the various ways of measuring performance complexity with Python.

One of the simplest time measurements can be done by using the `time` command of a POSIX/Linux system.

This is done by using the following command line:

```
$ time <command>
```

For example, here is a screenshot of the time it takes to fetch a very popular page from the Web:

```
anand@ubuntu-pro-book: /home/user/programs
File  Edit  View  Search  Terminal  Help
$ time wget http://www.google.com -O /dev/null
--2016-12-28 16:59:27--  http://www.google.com/
Resolving www.google.com (www.google.com)... 216.58.197.36, 2404:6800:4007:807::2004
Connecting to www.google.com (www.google.com)|216.58.197.36|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: http://www.google.co.in/?gfe_rd=cr&ei=F6JjWITaLeeK8Qfvj7OQBg [following]
--2016-12-28 16:59:27--  http://www.google.co.in/?gfe_rd=cr&ei=F6JjWITaLeeK8Qfvj7OQBg
Resolving www.google.co.in (www.google.co.in)... 216.58.197.35, 2404:6800:4007:807::2003
Connecting to www.google.co.in (www.google.co.in)|216.58.197.35|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: '/dev/null'

/dev/null               [ <=>                ]  12.14K  --.-KB/s    in 0.004s

2016-12-28 16:59:27 (2.80 MB/s) - '/dev/null' saved [12429]


real    0m0.121s
user    0m0.004s
sys     0m0.008s
$
```

Output of the time command on fetching a web page from the Internet via wget

the next section.

Timing code using the timeit module

The `timeit` module in the Python standard library allows the programmer to measure the time taken to execute small code snippets. The code snippets can be a Python statement, an expression, or a function.

The simplest way to use the `timeit` module is to execute it as a module in the Python command line.

For example, here is timing data for some simple Python inline code measuring the performance of a list comprehension calculating squares of numbers in a range:

```
$ python3 -m timeit '[x*x for x in range(100)]'
100000 loops, best of 3: 5.5 usec per loop

$ python3 -m timeit '[x*x for x in range(1000)]'
10000 loops, best of 3: 56.5 usec per loop

$ python3 -m timeit '[x*x for x in range(10000)]'
1000 loops, best of 3: 623 usec per loop
```

The result shows the time taken for execution of the code snippet. When run on the command line, the `timeit` module automatically determines the number of cycles to run the code and also calculates the average time spent in a single execution.

Finding out time complexity – graphs

Is it possible to find out from these results what the time performance complexity of our function is? Let us try plotting it in a graph and see the results.

The `matplotlib` library is very useful in plotting graphs in Python for any type of input data. We just need the following simple piece of code for this to work:

```
import matplotlib.pyplot as plt

def plot(xdata, ydata):
    """ Plot a range of ydata (on y-axis) against xdata (on x-axis)
    """

    plt.plot(xdata, ydata)
    plt.show()
```

The preceding code gives you the following output:

```
This is our x data.
>>> xdata = [100, 200, 400, 800, 1000]
This is the corresponding y data.
>>> ydata = [117,483,1920,7823,12395]
>>> plot(xdata, ydata)
```

Measuring CPU time with timeit

The `Timer` module by default uses the `perf_counter` function of the `time` module as the default `timer` function. As mentioned earlier, this function returns the wall clock time spent to the maximum precision for small time durations, hence it will include any sleep time, time spent for I/O, and so on.

This can be made clear by adding a little sleep time to our test function as follows:

```
def test():
    """ Testing the common_items function using a given input size """

    sleep(0.01)
    common = common_items(a1, a2)
```

The preceding code will give you the following output:

```
>>> t=timeit.Timer('test()','from common_items import test,setup;
setup(100)')
>>> 1000000.0*t.timeit(number=100)/100
10545.260819926625
```

The time jumped by as much as 300 times since we are sleeping `0.01` seconds (10 milliseconds) upon every invocation, so the actual time spent on the code is now determined almost completely by the sleep time as the result shows `10545.260819926625` microseconds (or about 10 milliseconds).

Sometimes you may have such sleep times and other blocking/wait times but you want to measure only the actual CPU time taken by the function. To use this, the `Timer` object can be created using the `process_time` function of the time module as the `timer` function.

This can be done by passing in a `timer` argument when you create the `Timer` object:

```
>>> from time import process_time
>>> t=timeit.Timer('test()','from common_items import
test,setup;setup(100)', timer=process_time)
>>> 1000000.0*t.timeit(number=100)/100
345.22438
```


Deterministic profiling

Deterministic profiling means that all function calls, function returns, and exception events are monitored, and precise timings are made for the intervals between these events. Another type of profiling, namely **statistical profiling**, randomly samples the instruction pointer and deduces where time is being spent – but this may not be very accurate.

Python, being an interpreted language, already has a certain overhead in terms of metadata kept by the interpreter. Most deterministic profiling tools makes use of this information and hence only add very little extra processing overhead for most applications. Hence deterministic profiling in Python is not a very expensive operation.

Profiling with cProfile and profile

The `profile` and `cProfile` modules provide support for deterministic profiling in the Python standard library. The `profile` module is purely written in Python. The `cProfile` module is a C extension that mimics the interface of the `profile` module but adds lesser overhead to it when compared to profile.

Third-party profilers

The Python ecosystem comes with a plethora of third-party modules for solving most problems. This is true in the case of profilers as well. In this section, we will take a quick look at a few popular third-party profiler applications contributed by developers in the Python community.

Line profiler

Line profiler is a profiler application developed by Robert Kern for performing line by line profiling of Python applications. It is written in Cython, an optimizing static compiler for Python that reduces the overhead of profiling.

Line profiler can be installed via `pip` as follows:

```
$ pip3 install line_profiler
```


Memory profiler

Memory profiler is a profiler similar to line profiler in that it profiles Python code line by line. However, instead of profiling the time taken in each line of code, it profiles lines by memory consumption.

Memory profiler can be installed the same way as line profiler:

```
$ pip3 install memory_profiler
```

Once installed, memory for lines can be printed by decorating the function with the `@profile` decorator in a similar way to line profiler.

Pympler

Pympler is a tool that can be used to monitor and measure the memory usage of objects in a Python application. It works on both Python 2.x and 3.x. It can be installed using `pip` as follows:

```
$ pip3 install pympler
```


The documentation of pympler is rather lacking. However, it's well-known use is to track objects and print their actual memory usage via its `asizeof` module.

The following is our `sub_string` function modified to print the memory usage of the sequences dictionary (where it stores all the generated substrings):

Mutable containers – lists, dictionaries, and sets

Lists, dictionaries, and sets are the most popular and useful mutable containers in Python.

Lists are appropriate for object access via a known index. Dictionaries provide a near constant time look-up for objects with known keys. Sets are useful to keep groups of items while dropping duplicates and finding their difference, intersection, union, and so on in near linear time.

Let us look at each of these in turn.

Lists

Lists provide a near constant time O(1) order for the following operations:

- `get(index)` via the `[]` operator
- The `append(item)` via the `.append` method

However, lists perform badly `(O(n))` in the following cases:

- Seeking an item via the `in` operator
- Inserting at an index via the `.insert` method

A list is ideal in the following cases:

- If you need a mutable store to keep different types or classes of items (heterogeneous).
- If your search of objects involves getting the item by a known index.
- If you don't have a lot of lookups via searching the list (**item in list**).
- If any of your elements are non-hashable. Dictionaries and sets require their entries to be hashable. So in this case, you almost default to using a list.

If you have a huge list – of, say, more than 100,000 items – and you keep finding that you search it for elements via the `in` operator, you should replace it with a dictionary.

Similarly, if you find that you keep inserting to a list instead of appending to it most of the time, you can think of replacing the list with `deque` from the `collections` module.

Dictionaries

Dictionaries provide a constant time order for:

- Setting an item via a key
- Getting an item via a key
- Deleting an item via a key

However, dictionaries take slightly more memory than lists for the same data. A dictionary is useful in the following situations:

- You don't care about the insertion order of the elements
- You don't have duplicate elements in terms of keys

A dictionary is also ideal where you load a lot of data uniquely indexed by keys from a source (database or disk) in the beginning of the application and need quick access to them – in other words, a lot of random reads as against fewer writes or updates.

Sets

The usage scenario of sets lies somewhere between lists and dictionaries. Sets are in implementation closer to dictionaries in Python – since they are unordered, don't support duplicate elements, and provide near O(1) time access to items via keys. They are kind of similar to lists in that they support the pop operation (even if they don't allow index access!).

Sets are usually used in Python as intermediate data structures for processing other containers – for operations such as dropping duplicates, finding common items across two containers, and so on.

Since the order of set operations is exactly same as that of a dictionary, you can use them for most cases where a dictionary needs to be used, except that no value is associated to the key.

Examples include:

- Keeping heterogeneous, unordered data from another collection while dropping duplicates
- Processing intermediate data in an application for a specific purpose – such as finding common elements, combining unique elements across multiple containers, dropping duplicates, and so on

Immutable containers – tuples

Tuples are an immutable version of lists in Python. Since they are unchangeable after creation, they don't support any of the methods of list modification such as insert, append, and so on.

Tuples have the same time complexity as when using the index and search (via **item in tuple**) as lists. However, they take much less memory overhead when compared to lists; the interpreter optimizes them more as they are immutable.

Hence tuples can be used whenever there are use cases for reading, returning, or creating a container of data that is not going to be changed but requires iteration. Some examples are as follows:

- Row-wise data loaded from a data store that is going to have only read access. For example, results from a DB query, processed rows from reading a CSV file, and so on.
- Constant set of values that needs iteration over and over again. For example, a list of configuration parameters loaded from a configuration file.
- When returning more than one value from a function. In this case, unless one explicitly returns a list, Python always returns a tuple by default.
- When a mutable container needs to be a dictionary key. For example, when a list or set needs to be associated to a value as a dictionary key, the quick way is to convert it to a tuple.

High performance containers – the collections module

The collection module supplies high performance alternatives to the built-in default container types in Python, namely `list`, `set`, `dict`, and `tuple`.

We will briefly look at the following container types in the collections module:

- `deque`: Alternative to a list container supporting fast insertions and pops at either ends
- `defaultdict`: Sub-class of `dict` that provides factory functions for types to provide missing values
- `OrderedDict`: Sub-class of `dict` that remembers the order of insertion of keys
- `Counter`: Dict sub-class for keeping count and statistics of hashable types

- `Chainmap`: Class with a dictionary-like interface for keeping track of multiple mappings
- `namedtuple`: Type for creating tuple-like classes with named fields

deque

A deque or *double ended queue* is like a list but supports nearly constant (O(1)) time appends and pops from either side as opposed to a list, which has an O(n) cost for pops and inserts at the left.

Deques also support operations such as rotation for moving `k` elements from back to front and reverse with an average performance of O(k). This is often slightly faster than the similar operation in lists, which involves slicing and appending:

```
def rotate_seq1(seq1, n):
    """ Rotate a list left by n """
    # E.g: rotate([1,2,3,4,5], 2) => [4,5,1,2,3]

    k = len(seq1) - n
    return seq1[k:] + seq1[:k]

def rotate_seq2(seq1, n):
    """ Rotate a list left by n using deque """

    d = deque(seq1)
    d.rotate(n)
    return d
```

By a simple `timeit` measurement, you should find that deques have a slight performance edge over lists (about 10-15%), in the above example.

defaultdict

Default dicts are dict sub-classes that use type factories to provide default values to dictionary keys.

A common problem one encounters in Python when looping over a list of items and trying to increment a dictionary count is that there may not be any existing entry for the item.

OrderedDict

OrderedDict is a sub-class of dict that remembers the order of the insertion of entries. It kind of behaves as a dictionary and list hybrid. It behaves like a mapping type but also has list-like behavior in remembering the insertion order plus supporting methods such as `popitem` to remove the last or first entry.

Here is an example:

```
>>> cities = ['Jakarta','Delhi','Newyork','Bonn','Kolkata',
'Bangalore','Seoul']
>>> cities_dict = dict.fromkeys(cities)
>>> cities_dict
{'Kolkata': None, 'Newyork': None, 'Seoul': None, 'Jakarta': None,
'Delhi': None, 'Bonn': None, 'Bangalore': None}

# Ordered dictionary
>>> cities_odict = OrderedDict.fromkeys(cities)
>>> cities_odict
OrderedDict([('Jakarta', None), ('Delhi', None), ('Newyork', None),
('Bonn', None), ('Kolkata', None), ('Bangalore', None), ('Seoul',
None)])
>>> cities_odict.popitem()
('Seoul', None)
>>> cities_odict.popitem(last=False)
('Jakarta', None)
```

You can compare and contrast how the dictionary changes the order around and how the `OrdredDict` container keeps the original order.

This allows a few recipes using the `OrderedDict` container.

Among other tools, we discussed objgraph and pympler – the former as a visualization tool to find relations and references between objects, helping to explore memory leaks, and the latter as a tool to monitor and report the memory usage of objects in the code and provide summaries.

In the last section on Python containers, we looked at the best and worst use case scenarios of standard Python containers – such as list, dict, set, and tuple. We then studied high performance container classes in the collections module – `deque`, `defaultdict`, `OrderedDict`, `Counter`, `Chainmap`, and `namedtuple`, with examples and recipes for each. Specifically, we saw how to create an LRU cache very naturally using `OrderedDict`.

Towards the end of the chapter, we discussed a special data structure called the bloom filter, which is very useful as a probabilistic data structure to report true negatives with certainty and true positives within a pre-defined error rate.

In the next chapter, we will discuss a close cousin of performance, scalability, where we will look at the techniques of writing scalable applications and the details of writing scalable and concurrent programs in Python.

Probabilistic data structures – bloom filters

Before we conclude our discussion on the container data types in Python, let us take a look at an important probabilistic data structure named **Bloom Filter**. Bloom filter implementations in Python behave like containers, but they are probabilistic in nature.

A bloom filter is a sparse data structure that allows us to test for the presence of an element in the set. However, we can only positively be sure of whether an element is not there in the set – that is, we can assert only for true negatives. When a bloom filter tells us an element is there in the set, it might be there – in other words, there is a non-zero probability that the element may actually be missing.

Bloom filters are usually implemented as bit vectors. They work in a similar way to a Python dictionary in that they use hash functions. However, unlike dictionaries, bloom filters don't store the actual elements themselves. Also elements, once added, cannot be removed from a bloom filter.

Bloom filters are used when the amount of source data implies an unconventionally large amount of memory if we store all of it without hash collisions.

In Python, the `pybloom` package provides a simple bloom filter implementation (however, at the time of writing, it doesn't support Python 3.x, so the examples here are shown in Python 2.7.x):

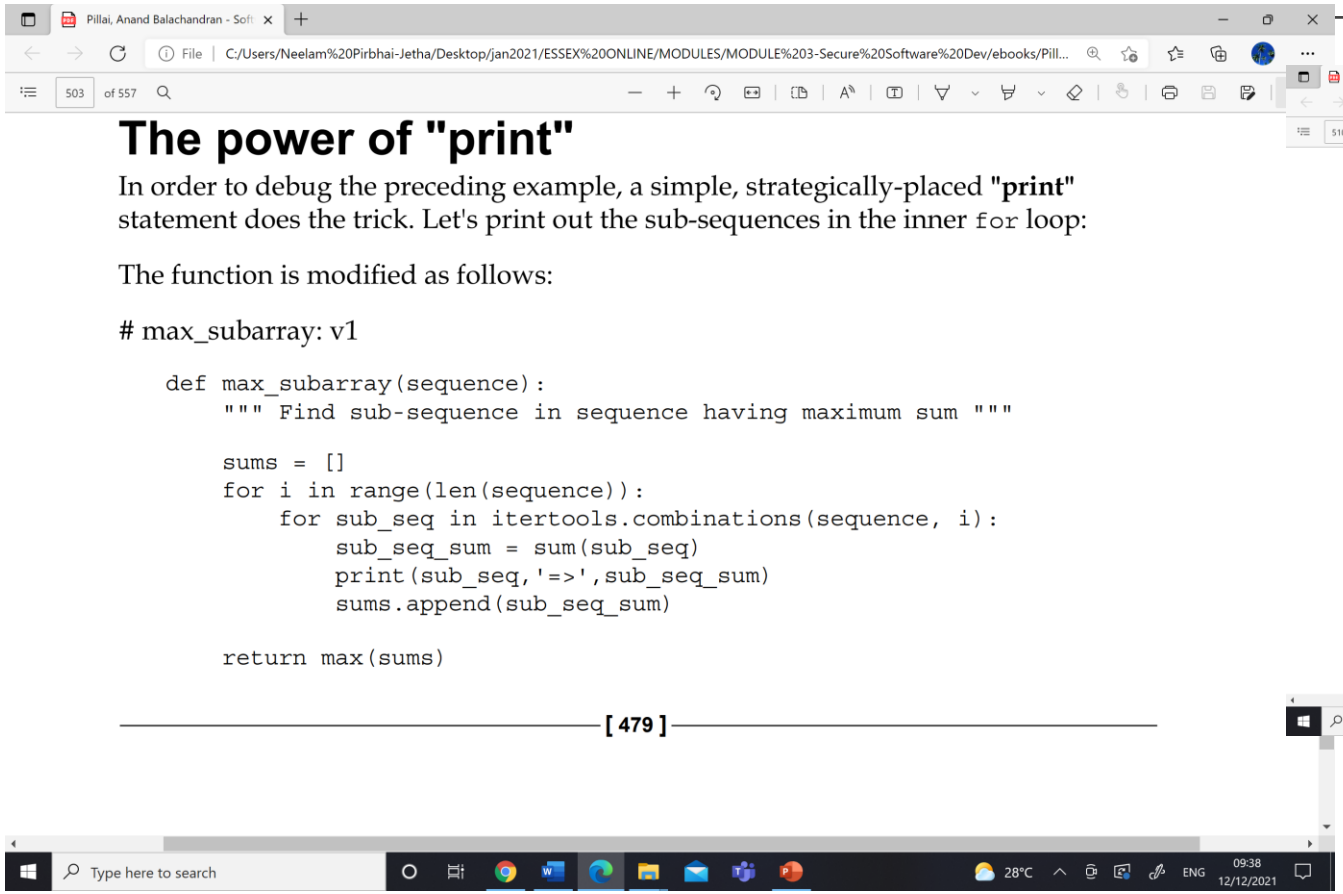
```
$ pip install pybloom
```

Let us write a program to read and index words from the text of `The Hound of Baskervilles`, which was the example we used in the discussion of the Counter data structure, but this time using a bloom filter:

```
# bloom_example.py
from pybloom import BloomFilter
import requests

f=BloomFilter(capacity=100000, error_rate=0.01)
```

Pillai (2018), Chapter 10, “Techniques for Debugging”



The power of "print"

In order to debug the preceding example, a simple, strategically-placed "**print**" statement does the trick. Let's print out the sub-sequences in the inner `for` loop:

The function is modified as follows:

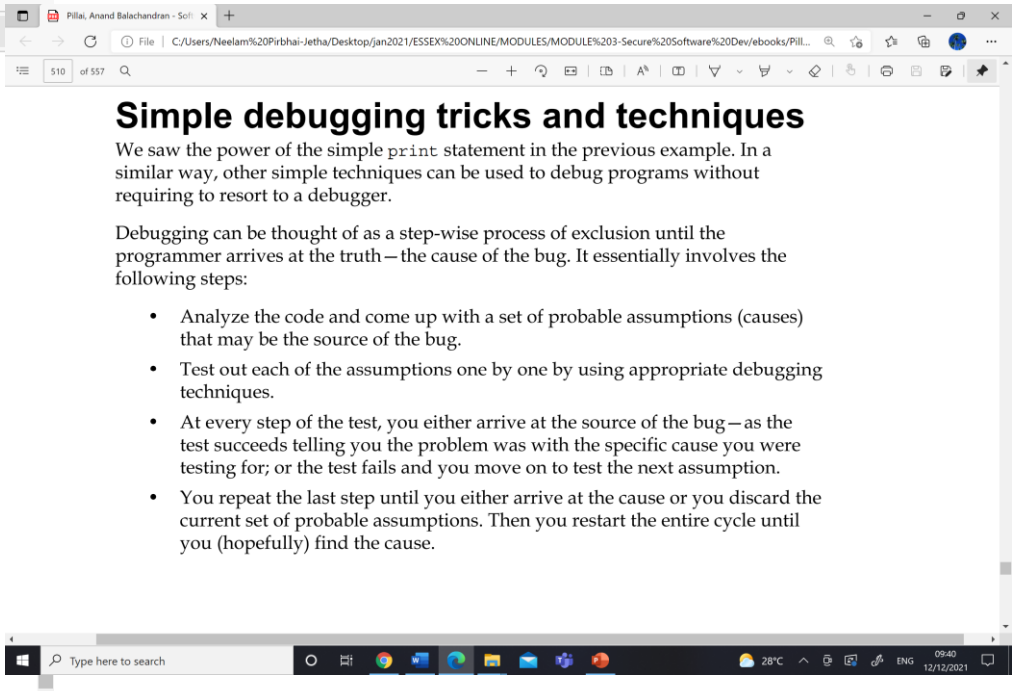
max_subarray: v1

```
def max_subarray(sequence):
    """ Find sub-sequence in sequence having maximum sum """

    sums = []
    for i in range(len(sequence)):
        for sub_seq in itertools.combinations(sequence, i):
            sub_seq_sum = sum(sub_seq)
            print(sub_seq,'=>',sub_seq_sum)
            sums.append(sub_seq_sum)

    return max(sums)
```

[479]



Simple debugging tricks and techniques

We saw the power of the simple `print` statement in the previous example. In a similar way, other simple techniques can be used to debug programs without requiring to resort to a debugger.

Debugging can be thought of as a step-wise process of exclusion until the programmer arrives at the truth—the cause of the bug. It essentially involves the following steps:

- Analyze the code and come up with a set of probable assumptions (causes) that may be the source of the bug.
- Test out each of the assumptions one by one by using appropriate debugging techniques.
- At every step of the test, you either arrive at the source of the bug—as the test succeeds telling you the problem was with the specific cause you were testing for; or the test fails and you move on to test the next assumption.
- You repeat the last step until you either arrive at the cause or you discard the current set of probable assumptions. Then you restart the entire cycle until you (hopefully) find the cause.

Pillai, Anand Balachandran - Soft

File | C:/Users/Neelam%20Pirbhai-Jetha/Desktop/jan2021/ESSEX%20ONLINE/MODULES/MODULE%203-Secure%20Software%20Dev/ebooks/Pill...

511 of 557

Word searcher program

In this section, we will look at some simple debugging techniques one by one using examples. We will start with the example of a word searcher program that looks for lines containing a specific word in a list of files—and appends and returns the lines in a list.

Here is the listing of the code for the word searcher program:

```
import os
import glob

def grep_word(word, filenames):
    """ Open the given files and look for a specific word.
    Append lines containing word to a list and
    return it """

    lines, words = [], []

    for filename in filenames:
        print('Processing',filename)
        lines += open(filename).readlines()


    word = word.lower()
    for line in lines:
        if word in line.lower():
            lines.append(line.strip())

    # Now sort the list according to length of lines
    return sorted(words, key=len)
```

Type here to search

28°C

Pillai, Anand Balachandran - Soft

File | C:/Users/Neelam%20Pirbhai-Jetha/Desktop/jan2021/ESSEX%20ONLINE/MODULES/MODULE%203-Secure%20Software%20Dev/ebooks/Pill...

515 of 557

Skipping blocks of code

A programmer can skip code blocks that they suspect of causing a bug during debugging. If the block is inside a loop, this can be done by skipping execution with a `continue` statement. We've seen an example of this already.

If the block is outside of a loop, this can be done by using an `if 0`, and moving the suspect code to the dependent block, as follows:

```
if 0:
    # Suspected code block
    perform_suspect_operation1(args1, args2, ...)
    perform_suspect_operation2(...)
```

If the bug disappears after this, then you're sure that the problem lies in the suspected blocks of code.

This trick has its own deficiency, in that it requires indenting large blocks of code to the right, which once the debugging is finished, should be indented back. Hence it is not advised for anything more than 5-6 lines of code.

Stopping execution

If you're in the middle of a hectic programming session, and you're trying to figure out an elusive bug, having already tried print statements, using the debugger, and other approaches, a rather drastic, but often fantastically useful, approach is to stop the execution just before or at the suspected code path using a function, `sys.exit` expression.

Type here to search

28°C ENG 09:41 12/12/2021

External dependencies—using wrappers

In cases where you suspect the problem is not inside your function, but in a function that you are calling from your code, this approach can be used.

Since the function is outside of your control, you can try and replace it with a wrapper function in a module where you have control.

For example, the following is generic code for processing serial JSON data. Let us assume that the programmer finds a bug with processing of certain data (maybe having a certain key-value pair), and suspects the external API to be the source of the bug. The bug may be that the API times out, returns a corrupt response, or in the worst case, causes a crash:

```
import external_api
def process_data(data):
    """ Process data using external API """

    # Clean up data—local function
    data = clean_up(data)
```


Saving to / loading data from files as cache

In this technique, you construct a filename using unique keys from the input data. If a matching file exists on disk, it is opened and the data is returned, otherwise, the call is made and the data is written. This can be achieved by using a *file caching* decorator as the following code illustrates:

```
import hashlib
import json
import os

def unique_key(address, site):
    """ Return a unique key for the given arguments """

    return hashlib.md5(''.join((address['name'],
                                address['street'],
                                address['city'],
                                site)).encode('utf-8')).hexdigest()

def filecache(func):
    """ A file caching decorator """

    def wrapper(*args, **kwargs):
        # Construct a unique cache filename
```

Logging as a debugging technique

Python comes with standard library support for logging via the aptly named `logging` module. Though print statements can be used as a quick and rudimentary tool for debugging, real-life debugging mostly requires that the system or application generate some logs. Logging is useful because of the following reasons:

- Logs are usually saved to specific log files, typically, with timestamps, and remain at the server for a while until they are rotated out. This makes debugging easy even if the programmer is debugging the issue some time after it happened.
- Logging can be done at different levels—from the basic INFO to the verbose DEBUG levels—changing the amount of information output by the application. This allows the programmer to debug at different levels of logging to extract the information they want, and figure out the problem.
- Custom loggers can be written, which can perform logging to various outputs. At its most basic, logging is done to log files, but one can also write loggers that write to sockets, HTTP streams, databases, and the like.

Debugging tools—using debuggers

Most programmers tend to think of *debugging* as something that they ought to do with a debugger. In this chapter, we have so far seen that more than an exact science, debugging is an art, which can be done using a lot of tricks and techniques rather than directly jumping to a debugger. However, sooner or later, we expected to encounter the debugger in this chapter—and here we are!

The Python Debugger, or pdb as it is known, is part of the Python runtime.

Pdb can be invoked when running a script from the beginning as follows:

```
$ python3 -m pdb script.py
```

However, the most common way in which programmers invoke pdb is to insert the following line at a place in the code where you want to enter the debugger:

```
import pdb; pdb.set_trace()
```

Let us use this, and try and debug an instance of the first example in this chapter, that is, the sum of the max subarray. We will debug the `O(n)` version of the code as an example:

```
def max_subarray(sequence):
    """ Maximum subarray - optimized version """
```

Summary

In this chapter, we learned about different debugging techniques with Python. We started with the simple *print* statement and followed it with simple tricks to debug a Python program such as using the *continue* statement in a loop, strategically placed `sys.exit` calls between code blocks, and so on.

We then looked at debugging techniques in some detail, especially on mocking and randomizing data. Techniques such as caching in files and in-memory database such as Redis were discussed with examples.

An example using Python schematics library showed generating random data for a hypothetical application in the healthcare domain.

The next section was about logging and using it as a debugging technique. We discussed simple logging using the *logging* module, advanced logging using `logger` object, and wrapped up the discussion by creating a logger wrapper with its custom formatting for logging time taken inside functions. We also studied an example of writing to syslog.

The end of the chapter was devoted to a discussion on debugging tools. You learned the basic commands of pdb, the Python debugger, and took a quick look at similar tools that provide a better experience, namely, iPdb and Pdb++. We ended the chapter with a brief discussion on tracing tools such as lptrace and the ubiquitous *strace* program on Linux.

This brings us to the conclusion of this chapter and the book.

To read the following…

Ferrer, J., Francisco, C. & Enrique, A. (2012) *Estimating Software Testing Complexity*. Information and Software Technology.

ISO/IEC/IEEE (2015) Software and Systems Engineering – Software Testing – Part 4: Test Techniques, ISO/IEC/IEEE 29119-4.

Shepperd, M. (1988) A Critique of Cyclomatic Complexity as a Software Metric, Software Engineering Journal.

Additional Reading

Song, I. Guedea-Elizalde, F. & Karray, F (2007) CONCORD: A Control Framework for Distributed Real-Time Systems. IEEE Sensors Journal 7(7):1078 – 1090.

Watson, A. H. & McCabe, T. J. (1996) Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. US Department of Commerce Technology Administration National Institute of Standards and Technology. Available from: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication500-235.pdf [Accessed 9 September 2021].

Caldeira, J., Brito e Abreu, F., Cardoso, J. & Reis, J. (2020) Unveiling Process Insights from Refactoring Practices. Computer Science, Arxiv.

Unit 6: Using Linters to Support Python Testing

NOTES

Definition: Lint/Linters

It is a tool that analyzes source code to flag programming errors, bugs, stylistic errors, and suspicious constructs. (cited from Wikipedia)

first linter was written by Stephen C. Johnson in 1978 while working in the Unix operating system at Bell Labs

Static Analysis means that automated software runs through your code source without executing it. It statically checks for potential bugs, memory leaks, and any other check that may be useful. *[Eg. Radon, Bandit for Python, Pep8*

Code standardizing

Security Issues [Brakeman – Static Analysis Security Tool, looks for SQL Injection

linters include a performance check

According to Ferit T., linting improves readability, removes silly errors before execution and code review. But, as mentioned, linting may do more complex jobs, like detecting code smells or performing static analysis of your codebase.

Source: https://sourcelevel.io/blog/what-is-a-linter-and-why-your-team-should-use-it

Reporting security flaws means the flaw can then quickly be remedied. Notifying the body concerned is called 'responsible disclosure'

References/Sources:

Government of the Netherlands (n.d.) Fighting Cybercrime in the Netherlands
The Computer Security Team (2020) Computer Security: Digital Stolen Goods of CERN?

Principles Of Secure Software Development (Source: Firdaus, A., Ghani, I. & Jeong, S. (2014) Secure Feature Driven Development (SFDD) Model for Secure Software Development, *Procedia Social and Behavioral Sciences* 129:546-553.)

A. The Principle of Least Privilege – Only the minimum necessary rights should be assigned to a subject that requests access to a resource and should be in effect for the shortest duration necessary. Other security activities that can be implement here is Minimize the damage, minimize interaction between privileged programs, password management, limit the access to database and restrict the access time.

B. The Principle of Failing Securely – When a system fails, it should do so securely. This behavior typically includes several elements: secure defaults the default is to deny access Security activities that can be implement here are Grant Access when not Explicitly forbidden, Ease of use, In case of mistake, access denied, No default passwords, No sample users, Files are write protected, owned by root, Error message generic, Error message information in log files.

C. c) The Principle of Securing the Weakest Link – Attackers are more likely to attack a weak spot in a software system than to penetrate a heavily fortified component. For example, some cryptographic algorithms can take many years to break, so attackers are unlikely to attack encrypted information communicated in a network

Cont…

D. The Principle of Defense in Depth – Layering security defenses in an application can reduce the chance of a successful attack. Incorporating redundant security mechanisms requires an attacker to circumvent each mechanism to gain access to a digital asset. For example, we need to use multiple layered protection software.

E. The Principle of Separation of Privilege – A system should ensure that multiple conditions are met before it grants permissions to an object. Checking access on only one condition may not be adequate for enforcing strong security. Compartmentalizing software into separate components that require multiple checks for access can inhibit an attack or potentially prevent an attacker from taking over an entire system.

F. The Principle of Economy of Mechanism – One factor in evaluating a system's security is its complexity. Keep design simple and remove unnecessary data and code.

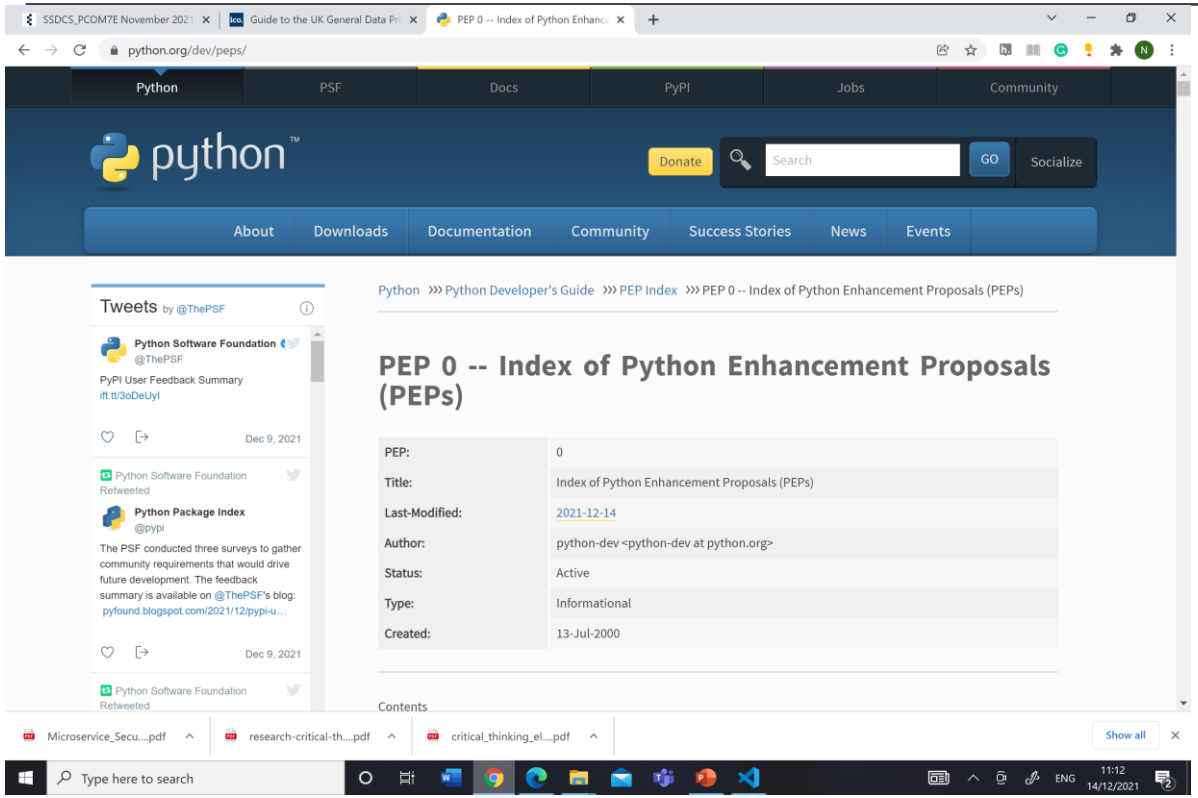
cont.

G. The Principle of Least Common Mechanism – Avoid having multiple subjects share those mechanisms that grant access to a resource. For example security activities can be add here is reduce potentially dangerous information flow, reduce possible interaction and make it more flexible.

H. The Principle of Complete Mediation – A software system that requires access checks to an object each time a subject requests access, especially for security Critical objects, decreases the chances that the system will mistakenly give elevated permissions to that subject. For example we need to make identification of source action. Make sure user is talking to authentication program, Safe login, Window control+alt+delete, Secure interface, Input validation. Do not authenticate based on IP source, email sender can be forged, hidden fields, and safely load

Information Commisioner's Office (ICO) (n.d) Guide to the General Data Protection Regulation (GDPR) https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/

Python.org (2020) PEP 0 -- Index of Python Enhancement Proposals (PEPs): Linters. https://www.python.org/dev/peps/



Mannino, J. (n.d.) Security in a Microservice World, OWASP, https://owasp.org/www-pdf-archive/Microservice_Security.pdf