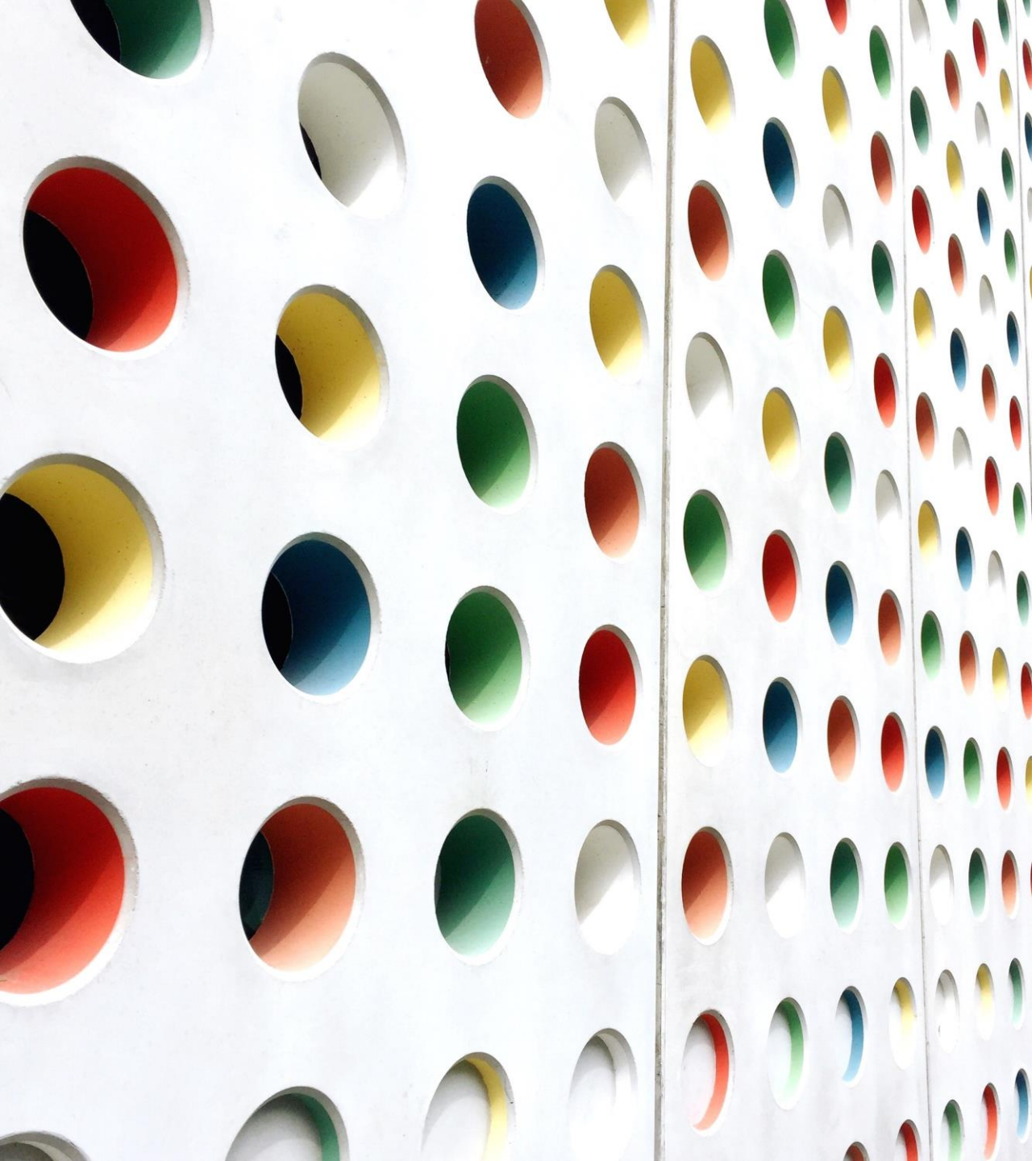


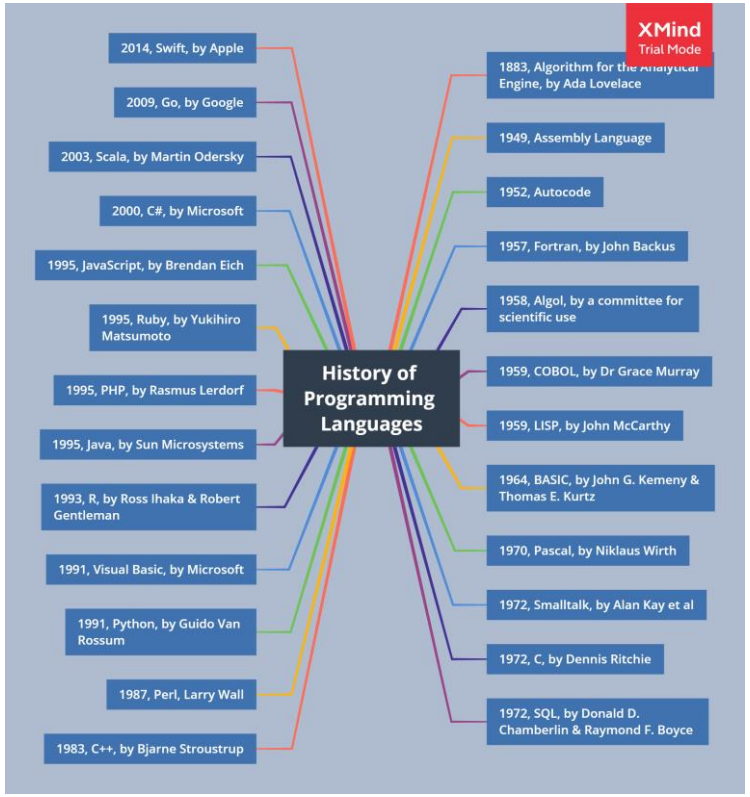
Unit 3: Programming Languages: History, Concepts & Design & Unit 4: Exploring Programming Language Concepts

Unit 3: Programming Languages: History, Concepts & Design

Programming Languages

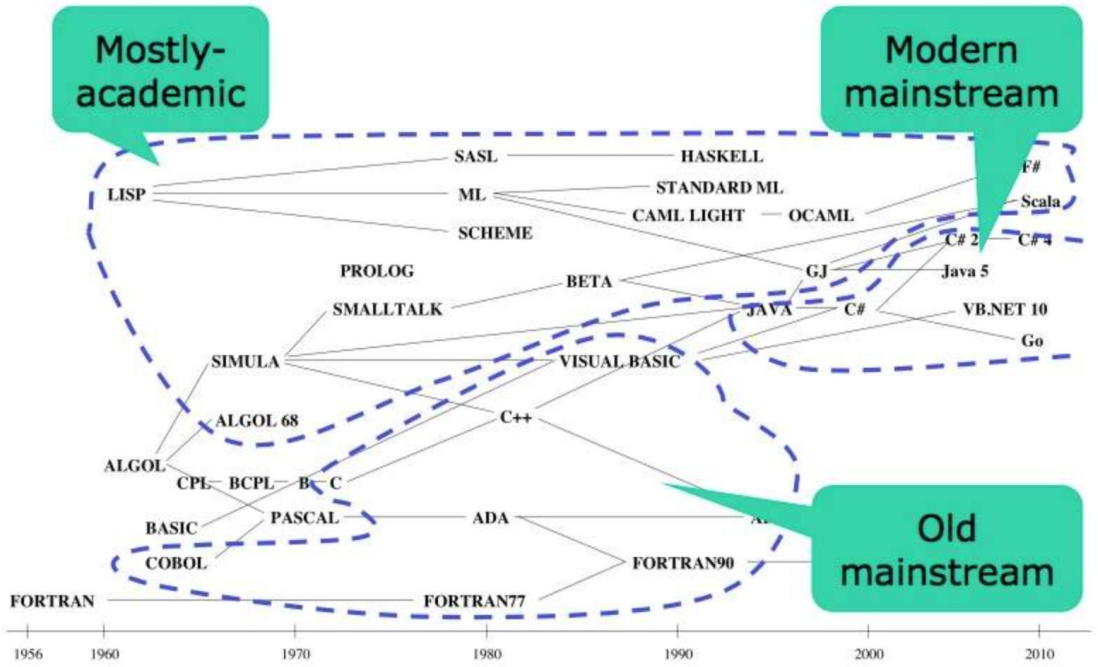
"Since developers can always find some slight flaw in any existing language, we are bound to witness the creation of new languages until the end of time". [Shane Markstrum, *Staking Claims: A History of Programming Language Design Claims and Evidence A Positional Work in Progress*]

History of Programming Languages



Content Source: https://www.onlinecollegeplan.com/computer-programming-languages/

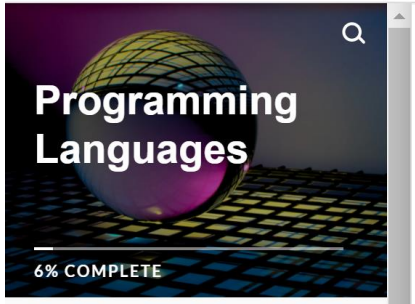
The second graphic is from Sestoft (2020) (below) which classifies languages into academic, old mainstream or modern mainstream. It also covers a much wider time period, almost up to the present day (2020). It also exhibits a more Eurocentric focus on the events covered. Nonetheless, it still omits a number of key events in the evolution of programming languages.



(Sestoft, 2020)

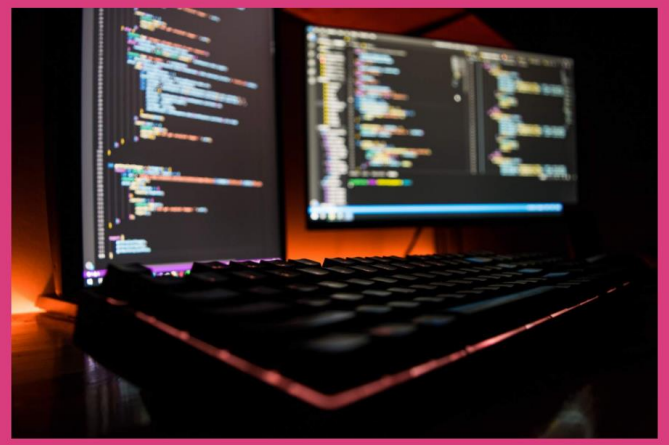
Course: Secure Software Develop | Programming Languages - Comp

my-course.co.uk/Computing/Computer%20Science/SSDCS/SSDCS%20Lecturecast%202/content/index.html#/lessons/LCQ2I5AEf4ztjM-mNUWtdB7ggRV...



Lesson 2 of 16

Compiled vs. Interpreted



A programming language is an abstraction designed to make it easier for humans to create programs to run on computing devices. However, for the computer to understand these programs they must first be converted into native machine code.

Introduction to Pr....pdf

Show all

Type here to search

28°C Mostly cloudy

ENG

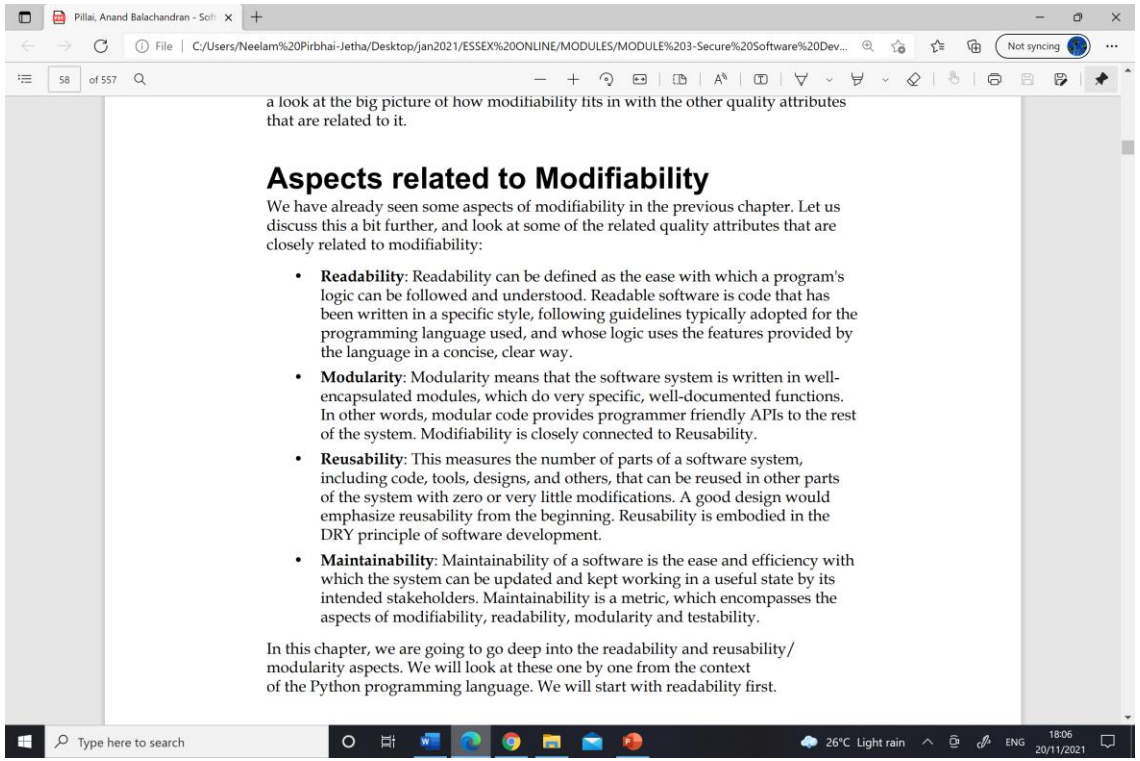
11:39
20/11/2021

Programming Paradigms

Van Roy (2009) defines a programming paradigm as 'an approach to programming a computer based on a mathematical theory or a coherent set of principles'.

Chapter 2,6,7,8 of the course text (Pillai, 2017)

Writing Modifiable and Readable Code



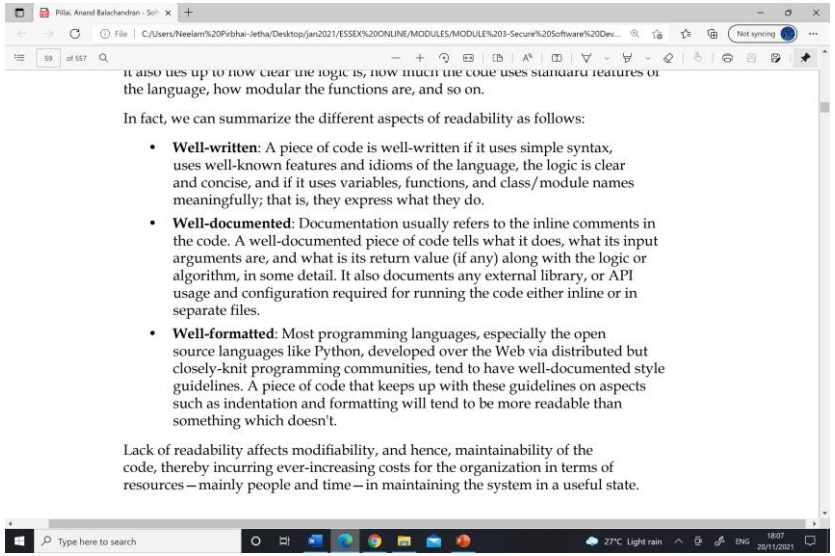
a look at the big picture of how modifiability fits in with the other quality attributes that are related to it.

Aspects related to Modifiability

We have already seen some aspects of modifiability in the previous chapter. Let us discuss this a bit further, and look at some of the related quality attributes that are closely related to modifiability:

- **Readability**: Readability can be defined as the ease with which a program's logic can be followed and understood. Readable software is code that has been written in a specific style, following guidelines typically adopted for the programming language used, and whose logic uses the features provided by the language in a concise, clear way.
- **Modularity**: Modularity means that the software system is written in well-encapsulated modules, which do very specific, well-documented functions. In other words, modular code provides programmer friendly APIs to the rest of the system. Modifiability is closely connected to Reusability.
- **Reusability**: This measures the number of parts of a software system, including code, tools, designs, and others, that can be reused in other parts of the system with zero or very little modifications. A good design would emphasize reusability from the beginning. Reusability is embodied in the DRY principle of software development.
- **Maintainability**: Maintainability of a software is the ease and efficiency with which the system can be updated and kept working in a useful state by its intended stakeholders. Maintainability is a metric, which encompasses the aspects of modifiability, readability, modularity and testability.

In this chapter, we are going to go deep into the readability and reusability/ modularity aspects. We will look at these one by one from the context of the Python programming language. We will start with readability first.

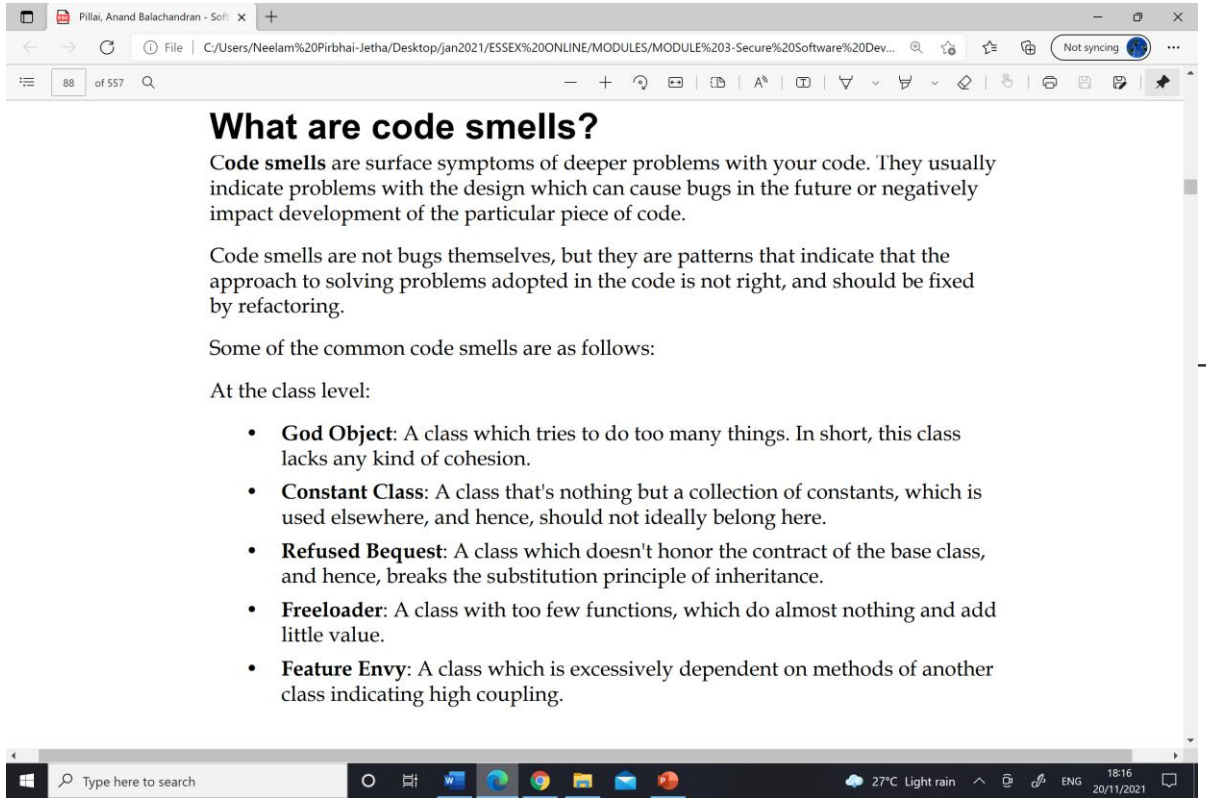


the language, how modular the functions are, and so on.

In fact, we can summarize the different aspects of readability as follows:

- **Well-written**: A piece of code is well-written if it uses simple syntax, uses well-known features and idioms of the language, the logic is clear and concise, and if it uses variables, functions, and class/module names meaningfully; that is, they express what they do.
- **Well-documented**: Documentation usually refers to the inline comments in the code. A well-documented piece of code tells what it does, what its input arguments are, and what is its return value (if any) along with the logic or algorithm, in some detail. It also documents any external library, or API usage and configuration required for running the code either inline or in separate files.
- **Well-formatted**: Most programming languages, especially the open source languages like Python, developed over the Web via distributed but closely-knit programming communities, tend to have well-documented style guidelines. A piece of code that keeps up with these guidelines on aspects such as indentation and formatting will tend to be more readable than something which doesn't.

Lack of readability affects modifiability, and hence, maintainability of the code, thereby incurring ever-increasing costs for the organization in terms of resources—mainly people and time—in maintaining the system in a useful state.



What are code smells?

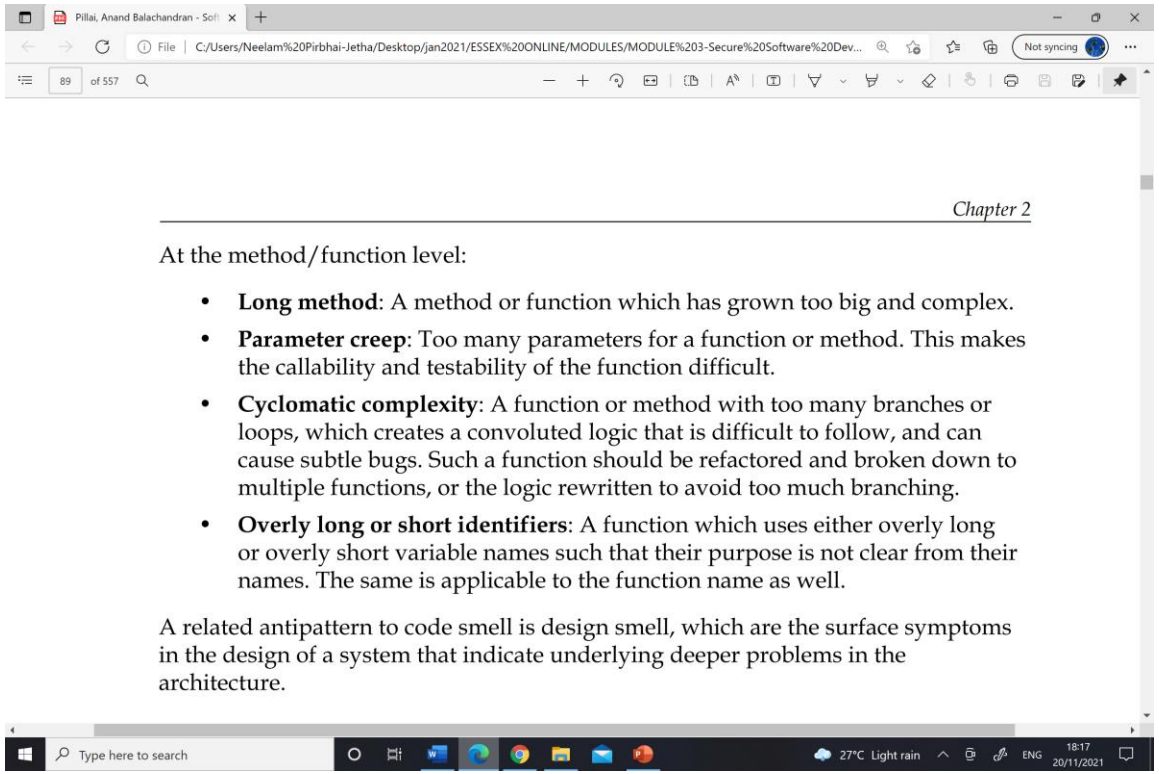
Code smells are surface symptoms of deeper problems with your code. They usually indicate problems with the design which can cause bugs in the future or negatively impact development of the particular piece of code.

Code smells are not bugs themselves, but they are patterns that indicate that the approach to solving problems adopted in the code is not right, and should be fixed by refactoring.

Some of the common code smells are as follows:

At the class level:

- **God Object**: A class which tries to do too many things. In short, this class lacks any kind of cohesion.
- **Constant Class**: A class that's nothing but a collection of constants, which is used elsewhere, and hence, should not ideally belong here.
- **Refused Bequest**: A class which doesn't honor the contract of the base class, and hence, breaks the substitution principle of inheritance.
- **Freeloader**: A class with too few functions, which do almost nothing and add little value.
- **Feature Envy**: A class which is excessively dependent on methods of another class indicating high coupling.



At the method/function level:

- **Long method**: A method or function which has grown too big and complex.
- **Parameter creep**: Too many parameters for a function or method. This makes the callability and testability of the function difficult.
- **Cyclomatic complexity**: A function or method with too many branches or loops, which creates a convoluted logic that is difficult to follow, and can cause subtle bugs. Such a function should be refactored and broken down to multiple functions, or the logic rewritten to avoid too much branching.
- **Overly long or short identifiers**: A function which uses either overly long or overly short variable names such that their purpose is not clear from their names. The same is applicable to the function name as well.

A related antipattern to code smell is design smell, which are the surface symptoms in the design of a system that indicate underlying deeper problems in the architecture.

Summary

In this chapter, we looked at the architectural quality attribute of modifiability, and its various aspects. We discussed readability in some detail, including the readability antipatterns along with a few coding antipatterns.

We looked at various techniques for improving readability of code and understood the different aspects of commenting of code such as function, class and module docstrings. We also looked at PEP-8, the official coding convention guideline for Python.

We then looked at some rules of thumb for code comments, and went on to discuss the fundamentals of modifiability, namely, coupling and cohesion of code. We looked at different cases of coupling and cohesion with a few examples. We then went on to discuss the strategies of improving modifiability of code such as providing explicit interfaces or APIs, avoiding two-way dependencies, abstracting common services to helper modules, and using inheritance techniques. We looked at an example where we refactored a class hierarchy via inheritance to abstract away common code and to improve the modifiability of the system.

Towards the end, we listed the different tools providing static code metrics in Python such as PyLint, Flake8, PyFlakes, and others. We learned about McCabe Cyclomatic complexity with the help of a few examples. We also learned what code smells are, and performed a refactoring exercise to improve the quality of the piece of code in stages.

In the next chapter, we'll discuss another important quality attribute of software architecture, namely, Testability.

7
Design Patterns in Python

Design Patterns simplify building software by reusing successful designs and architectures. Patterns build on the collective experience of software engineers and architects. When encountered with a problem which needs new code to be written, an experienced software architect tends to make use of the rich ecosystem of available design/architecture patterns.

Patterns evolve when a specific design proves successful in solving certain classes of problems repeatedly. When experts find that a specific design or architecture helps them to solve classes of related problems consistently, they tend to apply it more and more, codifying the structure of the solution into a pattern.

Python, being a language which supports dynamic types, and high-level object oriented structures like classes and metaclasses, first-class functions, co-routines, callable objects, and so on, is a very rich playground for constructing reusable design and architecture patterns. In fact, as opposed to languages like C++ or Java, you would often find there are multiple ways of implementing a specific design pattern in Python. Also, more often than not, you would find that the Pythonic ways of implementing a pattern is more intuitive and illustrative than, say, copying a standard implementation from C++/Java into Python.

This chapter's focus is mostly on this latter aspect—illustrating how one can build design patterns which are more Pythonic than what usual books and literature on this topic tend to do. It doesn't aim to be a comprehensive guide to design patterns, though we would be covering most of the usual aspects as we head into the content.

The topics we plan to cover in this chapter are as follows:

- Design patterns elements
- Categories of design patterns
- Pluggable hashing algorithms
- Summing up pluggable hashing algorithms

- Patterns in Python – Creational
  - The Singleton pattern
  - The Borg pattern
  - The Factory pattern
  - The Prototype pattern
  - The Builder pattern
- Patterns in Python – Structural
  - The Adapter pattern
  - The Facade pattern
  - The Proxy pattern
- Patterns in Python – Behavioral
  - The Iterator pattern
  - The Observer pattern
  - The State pattern

Software Security and Secure Coding

Software security and secure coding has assumed more importance than ever due to the unprecedented amounts of data being shared across software and hardware systems–the explosion of smart personal technologies such as smart phones, smart watches, smart music players, and other smart systems has aided this immense traffic of data across the Internet in a big way. With the advent of IPV6 and expected large scale adoption of IoT devices (Internet of Things) in the next few years, the amount of data is only going to increase exponentially. P. 281

Unit 4: Exploring Programming Language Concepts

Jaiswal, S. (2020) Python Regular Expression Tutorial. Available from: https://www.datacamp.com/community/tutorials/python-regular-expression-tutorial

---

Regular Expressions, often shortened as **regex**, are a sequence of characters used to check whether a pattern exists in a given text (string) or not.

Regular Expressions:

- are **used** at the server side to validate the format of email addresses or passwords during registration,
- are used for parsing text data files to find, replace, or delete certain string, etc.
- help in manipulating textual data, which is often a prerequisite for data science projects involving text mining.

Regex in Python

In Python, regular expressions are supported by the **re** module.

```
import re
```

To read the article for the different features and how to use regex in Python

Summary of the features in the next two tables:

Character(s)	What it does
.	A period. Matches any single character except the newline character.
^	A caret. Matches a pattern at the start of the string.
\A	Uppercase A. Matches only at the start of the string.
$	Dollar sign. Matches the end of the string.
\Z	Uppercase Z. Matches only at the end of the string.
[ ]	Matches the set of characters you specify within it.
\	· If the character following the backslash is a recognized escape character, then the special meaning of the term is taken. · Else the backslash () is treated like any other character and passed through. · It can be used in front of all the metacharacters to remove their special meaning.
\w	Lowercase w. Matches any single letter, digit, or underscore.
\W	Uppercase W. Matches any character not part of \w (lowercase w).
\s	Lowercase s. Matches a single whitespace character like: space, newline, tab, return.
\S	Uppercase S. Matches any character not part of \s (lowercase s).
\d	Lowercase d. Matches decimal digit 0-9.
\D	Uppercase D. Matches any character that is not a decimal digit.

\t	**Lowercase t. Matches tab.**
\n	Lowercase n. Matches newline.
\r	Lowercase r. Matches return.
\b	Lowercase b. Matches only the beginning or end of the word.
+	Checks if the preceding character appears one or more times.
*	Checks if the preceding character appears zero or more times.
?	· Checks if the preceding character appears exactly zero or one time. · Specifies a non-greedy version of +, *
{ }	Checks for an explicit number of times.
( )	Creates a group when performing matches.
< >	Creates a named group when performing matches.

Larson, E. (2016) Generating Evil Test Strings for Regular Expressions. IEEE International Conference on Software Testing, Verification and Validation (ICST).

Regular expressions are a powerful string processing tool. However, they are error-prone and receive little error checking from the compiler as most regular expressions are syntactically correct. This paper describes EGRET, a tool for generating evil test strings for regular expressions. EGRET focuses on common mistakes made by developers when creating regular expressions and develops test strings that expose these errors.

Despite their widespread use, regular expressions are error-prone. **First**, **the "language"** used to specify regular expressions is designed to be compact, using **punctuation marks** to represent **different operations**, allowing complex regular expressions to be written succinctly.

Second, when the regular expression is **compiled at run-time**, only **limited error-checking related to syntax is done**. Most regular expressions are free of syntax errors.

The EGRET (Evil Generation of Regular Expression Tests) tool.

EGRET contains a command-line interface and a graphical web interface. They both operate in a similar fashion. A user enters a regular expression and receives a list of accepted string and rejected strings. EGRET may also emit error and warning messages in certain situations. An optional feature allows users to see the contents of any groups for an accepted string. The web interface also allows the user to enter their own test string and determine if the string is accepted or rejected. The command-line interface also supports file input/output.

Sources:

Larson, E. (2016) Generating Evil Test Strings for Regular Expressions. IEEE International Conference on Software Testing, Verification and Validation (ICST).
Larson, E. (2018) Automatic Checking of Regular Expressions. 18th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)

ACRE is inspired by a variety of **lint tools** including LCLint [7], Splint [8], PCLint [10], DLint [11], and Android Lint [12].

Like ACRE, these **lint tools employ a variety of checks**, most of them are relatively simple, that **detect likely programming mistakes**.

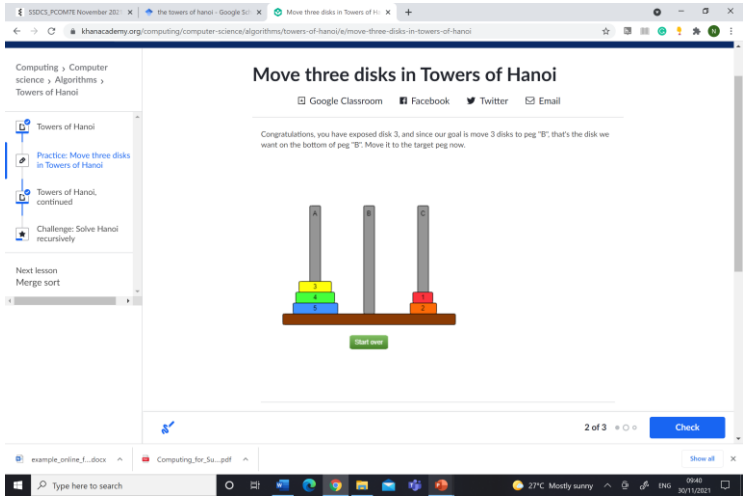
Test generation is a popular approach to finding bugs in regular expressions. EGRET [17] generates evil test strings given a regular expression.

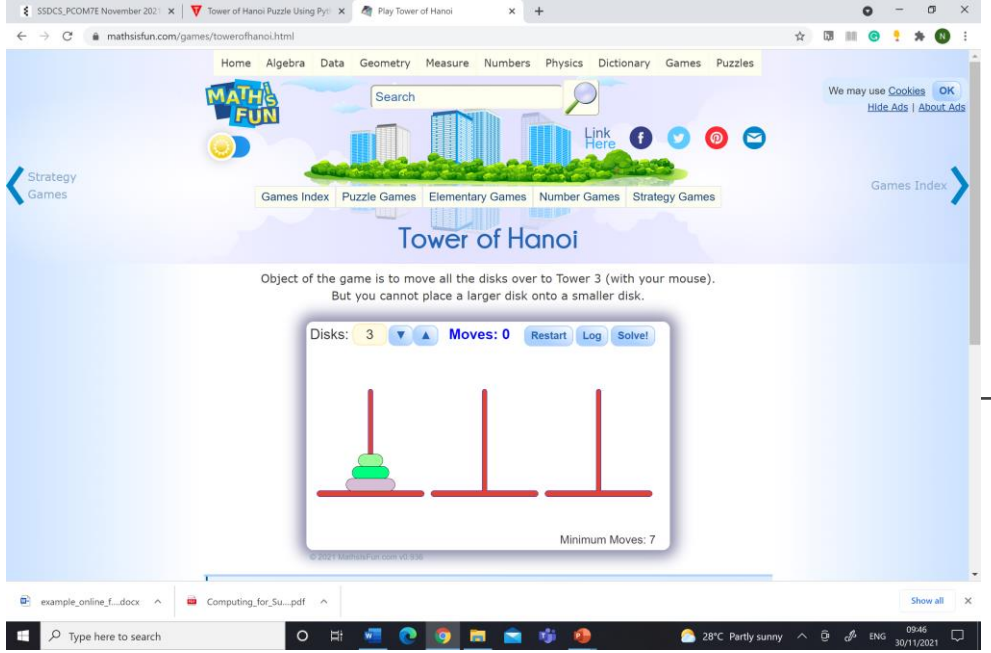
the EGRET (Evil Generation of Regular Expression Tests) tool. EGRET takes a regular expression as input and generates test strings. The "evil" aspect of EGRET focuses on generating test strings to expose common errors made by programmers. Some strings generated by EGRET are actually intended to be rejected by the regular expression. EGRET creates two lists of test strings: accepted strings and rejected strings. A user can quickly scan both of these lists and identify strings that are incorrectly classified. Even if no bugs are found, the user has higher confidence that the regular expression is working as intended.

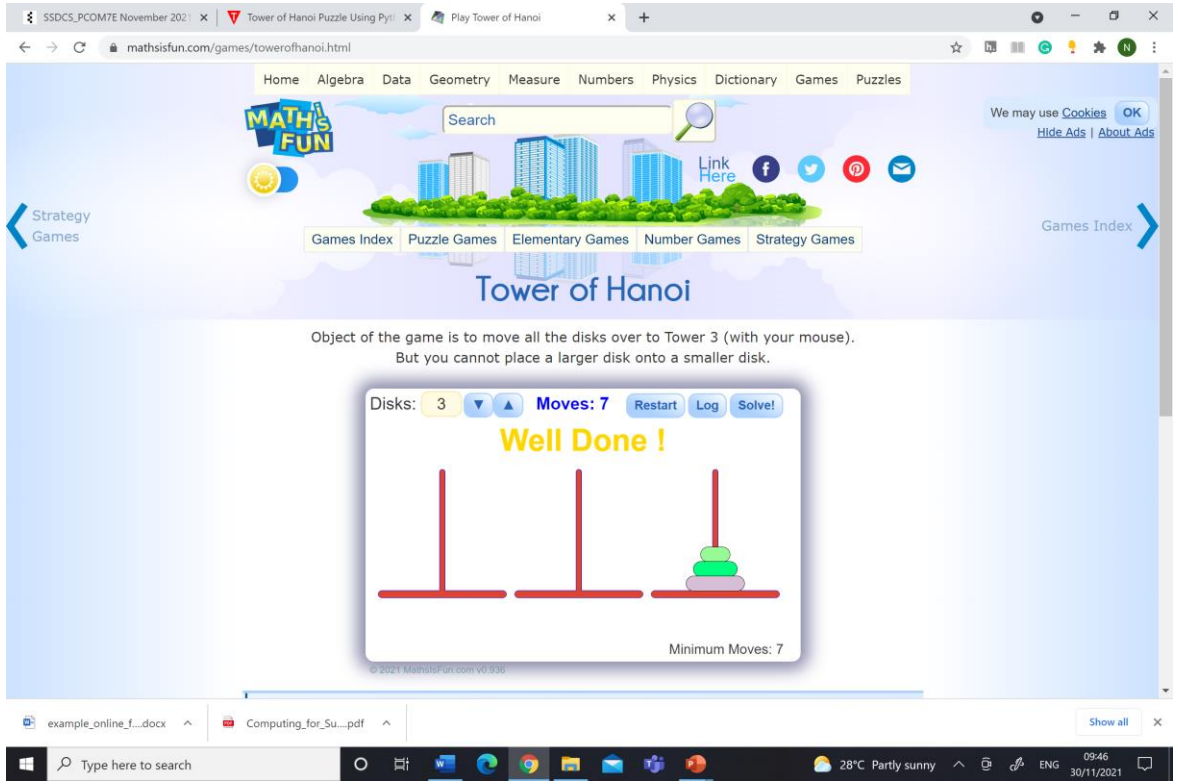
Recursion
Source: https://www.khanacademy.org/computing/computer-science/algorithms/recursive-algorithms/a/recursion

---

What do Russian dolls have to do with algorithms? Just as one Russian doll has within it a smaller Russian doll, which has an even smaller Russian doll within it, all the way down to a tiny Russian doll that is too small to contain another, we'll see **how to design an algorithm to solve a problem by solving a smaller instance of the same problem**, unless the problem is so small that we can just solve it directly. We call this technique **recursion**.




mathsisfun.com/games/towerofhanoi.html
Home Algebra Data Geometry Measure Numbers Physics Dictionary Games Puzzles
Search
We may use Cookies OK
Hide Ads | About Ads
Link Here
Strategy Games
Games Index
Games Index Puzzle Games Elementary Games Number Games Strategy Games
Tower of Hanoi
Object of the game is to move all the disks over to Tower 3 (with your mouse).
But you cannot place a larger disk onto a smaller disk.
Disks: 3
Moves: 0
Restart Log Solve!
Minimum Moves: 7
example_online_f....docx
Computing_for_Su....pdf
Show all
Type here to search
28°C Partly sunny
ENG
09:46
30/11/2021


mathsisfun.com/games/towerofhanoi.html
Home Algebra Data Geometry Measure Numbers Physics Dictionary Games Puzzles
Search
We may use Cookies OK
Hide Ads | About Ads
Link Here
Strategy Games
Games Index
Games Index Puzzle Games Elementary Games Number Games Strategy Games
Tower of Hanoi
Object of the game is to move all the disks over to Tower 3 (with your mouse).
But you cannot place a larger disk onto a smaller disk.
Disks: 3
Moves: 7
Restart Log Solve!
Well Done !
Minimum Moves: 7
example_online_f....docx
Computing_for_Su....pdf
Show all
Type here to search
28°C Partly sunny
ENG
09:46
30/11/2021

https://pythonschool.net/regular-expressions/regular-expressions/

Regular expressions provide a powerful method of **pattern matching** in text documents. They enable you to:

- **find** text that matches the given pattern
- **replace** text that matches the given pattern
- **validate** whether text matches a given pattern

- Regular expressions provide a way to escape from these limitations but this freedom comes at a cost - you have to learn regular expressions syntax.

Regular Expression	Meaning	Example Match
`a`	this is the most basic regular expression, it will match only the provided character	
`orange`	this regular expression will match the word orange	
`d.g`	a dot/period represents any character	e.g. dog, dig, dag, dxg
`harbou?r`	a question mark represents zero or one of the preceeding element	e.g. harbour, harbor
`a*`	an asterisk represents zero or more of the preceeding element	e.g. , a, aa, aaa, aaaa
`a+`	a plus represents one or more of the preceeding element	e.g. a, aa, aaa, aaaa
`uk|us`	a pipe represents one element **OR** the other	e.g. uk, us
`ja(b|m)`	brackets are used to group elements together	e.g. jab, jam

Further Regular Expression Syntax

by Adam McNicol.
30 August 2014.
updated on 30 August 2014.

Python | Regular Expressions | Regex

Whilst the syntax on the previous page will enable us to create regular expressions for some quite complex situations there are other symbols which simplify the writing of regular expressions in many situations. The table below introduces some of these:

Regular Expression	Meaning	Example Match
`bre[ae]d`	characters in square brackets are another way of representing **OR**	e.g. bread, breed
`[a-z]`	this matches any character in the range a to z (lower-case only)	e.g. a, z, q, r
`[a-zA-Z]`	this matches any character in the range a to z (lower and upper-case)	e.g. a, A, q, Q
`[a-zA-Z0-9]`	this matches any character between a-z (lower and upper-case) and the digits 0-9	e.g. a, A, q, Q, 1, 5
`^hello`	a caret means that the element after it must be found at the start of the string	e.g. , hello, hello world
`world$`	a dollar sign means the the preceding element must be found at the end of the string	e.g. world, hello world
`a{2}`	means match to precisely two instances of the character a	e.g. aa
`a{1,3}`	means match to a minimum of 1 and maximum of 3 instances of the character a	e.g. a, aa, aaa
`\.com`	a backslash is an escape character so that the dot/period is not taken to be a special character	e.g. .com
`d`	represents a digit i.e. 0-9	e.g. 0, 5, 7
`s`	represents a single space	

use are they actually in practice. Turns out they are used pretty much everywhere! Here are a few examples:

- Validating form input from a web page
- Rewriting URLs to a more user friendly format on a web server
- Making sure files of a particular type are handled correctly on a web server
- Find and replace in a text editor

```
server {
    #listen 80; ## listen for ipv4; this line is default and implied
    #listen [::]80 default_server ipv6only=on; ##listen for ipv6

    root /usr/share/nginx/html;
    index index.html index.htm index.php;

    #Make site accessible from http://localhost/
    #server_name localhost;
    server_name python-school-server;

    location / {
        #First attempt to server request as file, then
        #as directory, then fall back to displaying a 404.
        try_files $uri $uri/ =404;
        allow 192.168.1.0/24
        allow 127.0.0.1;
        deny all;
        #Uncomment to enable naxsi on this location
        #include /etc/nginx/naxsi.rules
    }

    location ~ \.php$ {
        try_files $uri $uri/ =404;
        allow 192.168.1.0/24
        allow 127.0.0.1;
        deny all;
        include fastcgi_params;
        fastcgi_pass php5-fpm-sock;
    }
}
```

30 August 2014.
updated on 30 August 2014.

Python | Regular Expressions | Regex

Some key terms

Before we start using regular expressions in our Python code it is important that we recognise some of the key terms that are associated with the topic:

Key term	Explanation
Regex	Abbreviation of regular expression
Literal	Any character that is used in the regular expression e.g. a
Metacharacter	This is a character with special meaning. There are 13 of these: `| ? * + ( ) [ ] \ ^ $ . -`
Alternation	The pipe separating alternatives e.g. `uk|us`
Character class	A character class is any alternatives expressed using square brackets e.g. `[a-z]`
Grouping	Using round brackets to group element together to define the scope and precedence of operators e.g. `b(a|e)d`
Quantification	Using a quantifier (`e.g. ? * +`) to specify how many of the preceding element is allowed

Now that we have been introduced to the syntax and terminology surrounding regular expressions and we have created some of our own it is time to use them in a practical situation. Let's look at regular expressions and Python...

Previous - Further Regular Expression Syntax **Next** - Python and regular expressions