

Test Driven Development

Best Coding Practices

Krzysztof Buzar

DSR February 2024

Q: How many computers are there on a spaceship?

The Random Neutrino Problem

- Voting Machine changed outcome because of random collision with a cosmic particle
- Mario jumps in a way that the game did not plan for
- A plane almost crashed!

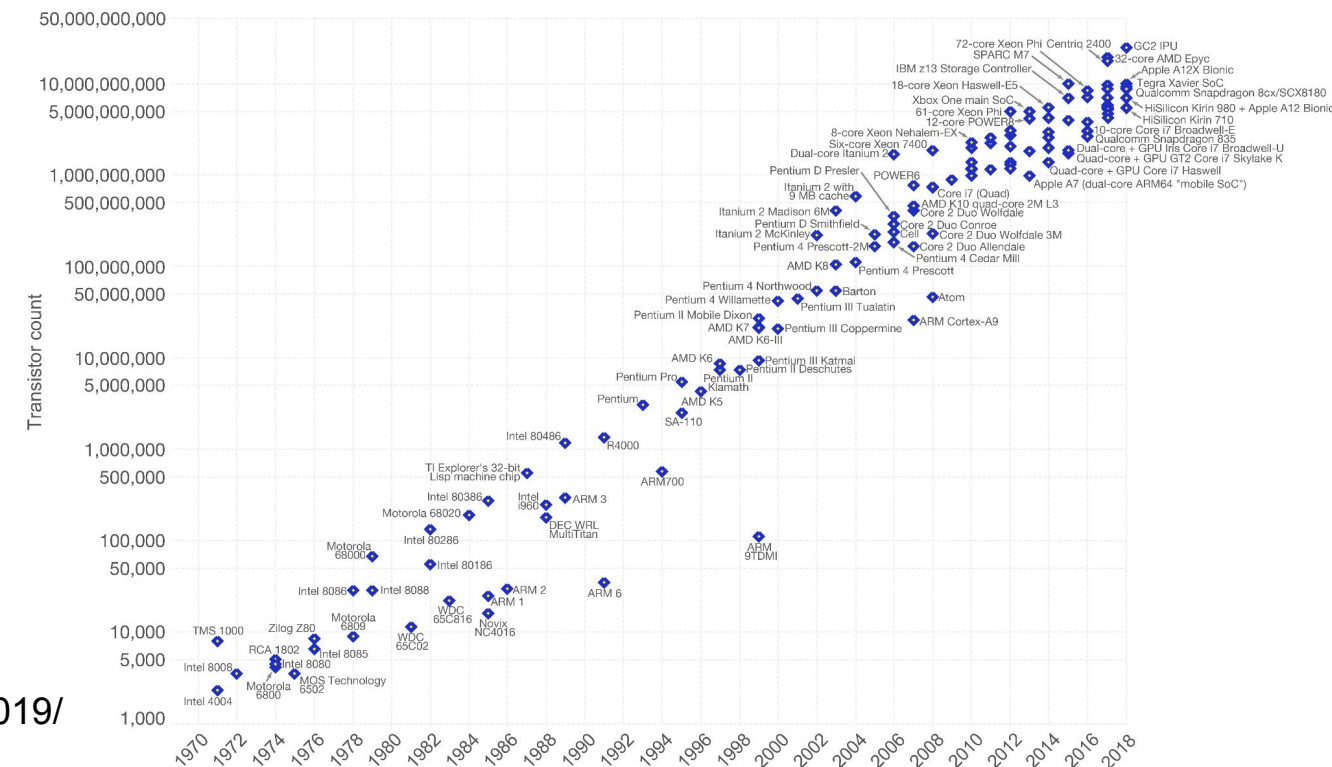
Endless Loops and Finite Resources

- Moore's Law and Quantum Computers

Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.

OurWorld
in Data



<https://www.visualcapitalist.com/visualizing-moores-law-in-action-1971-2019/>

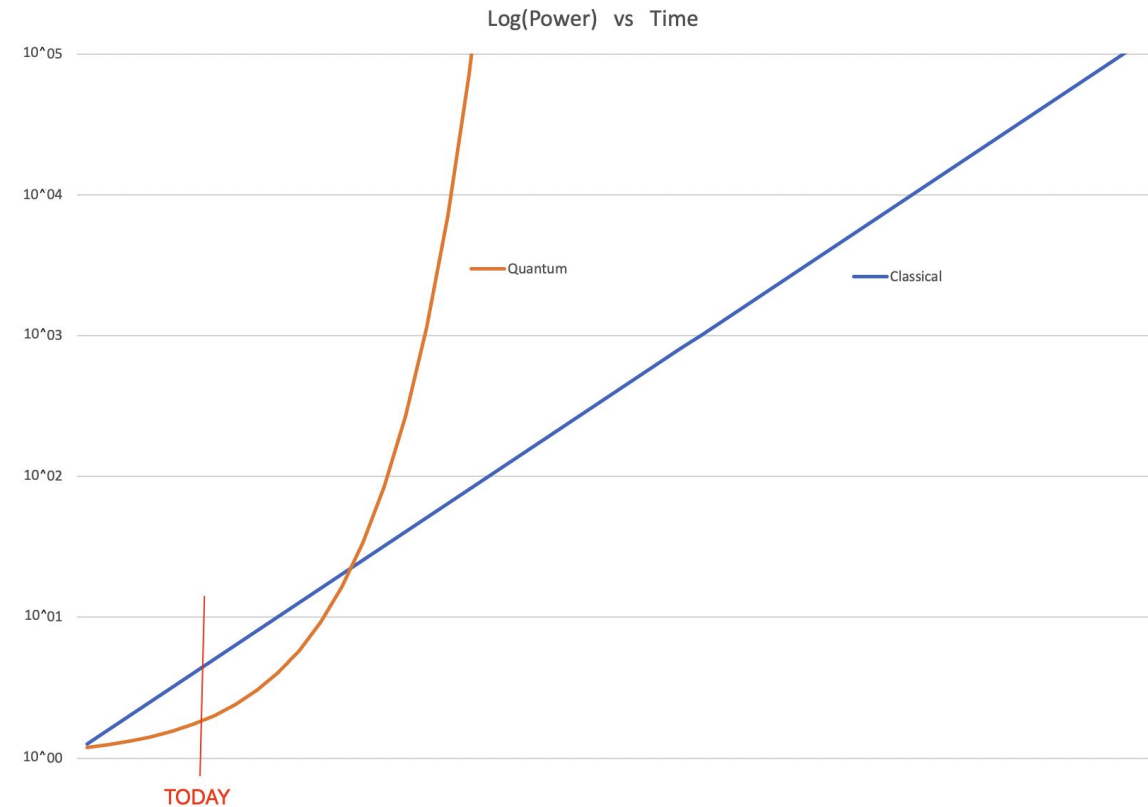
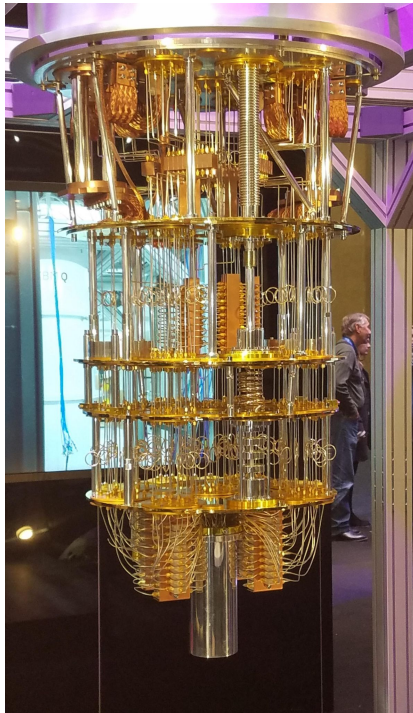
Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

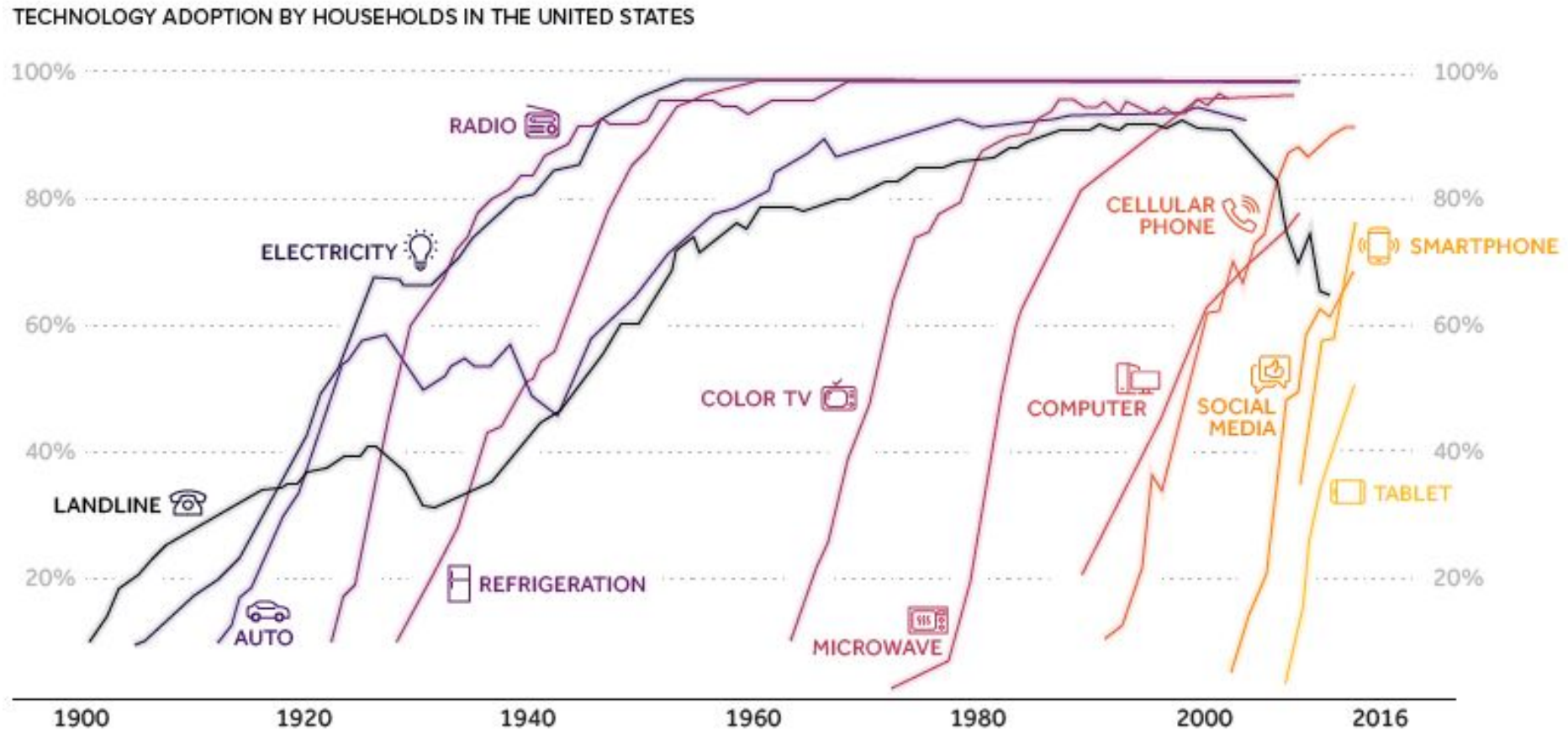
Licensed under CC-BY-SA by the author Max Roser.

Endless Loops and Finite Resources

- Quantum Computers (IBM Q)



Technology adaptation and range

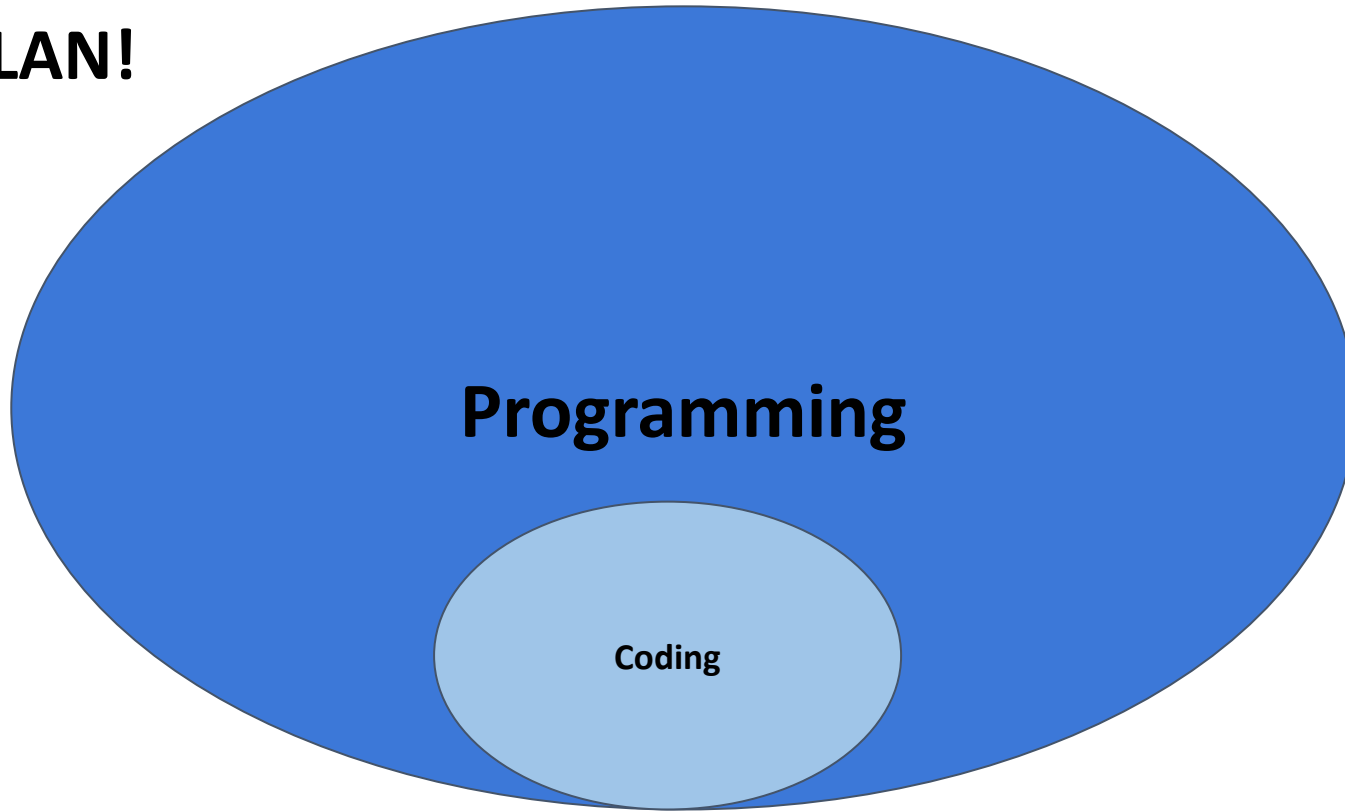


Coding and Programming

Q: What is the difference?

The most important part is...

HAVING A PLAN!



A: 'Physical activity of writing code'

Programming as a Concept

With many equally important parts!

- **Planning!**
- Problem Solving
- Design
- Collaboration
- Innovation
- Long Term Thinking
- Organization
- ... Actually writing some code! :D

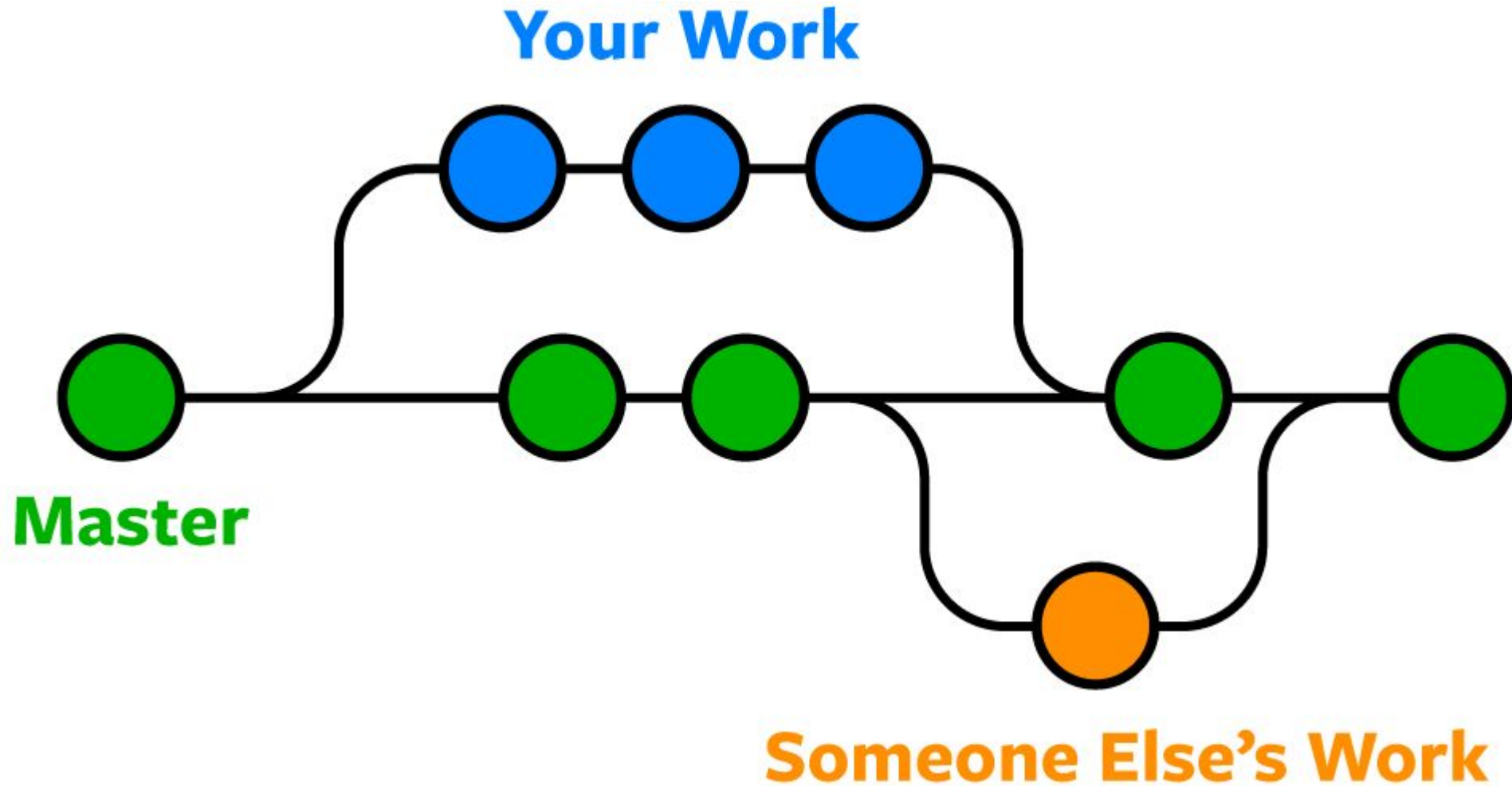
Agenda

1. Tests and why do we NEED them (for Planning!)
2. Repository Structure and Project Management
3. Git Crash Course
4. Best Coding Practices - Context with sources
5. Best Practices Overview

Practical aspects of Project Management

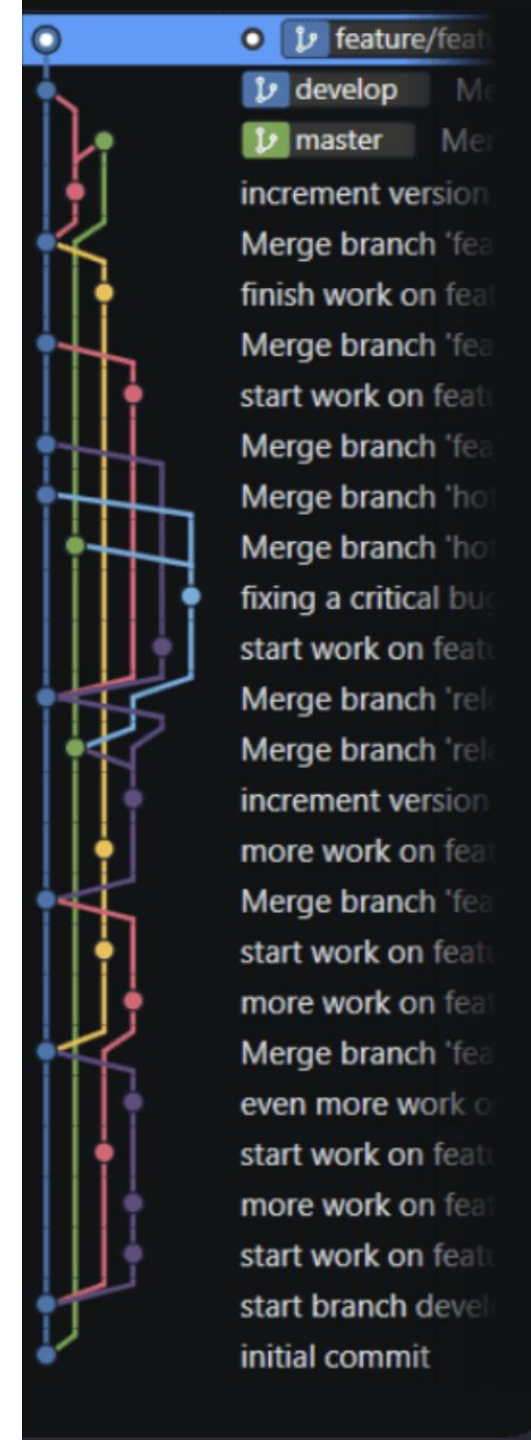
- Version Control
 - Git
 - Git GUIs
 - Gitlab
- Agile
 - Jira/Kanban/Scrum
 - Notion
- Project Structure from Scratch
 - [Example project structure](#)
 - Cookiecutter!

Version control



Version control

- Git visual APIs
 - Fork (Windows, Mac)
 - GitKraken (Ubuntu)
 - Great PyCharm Integration!



Git on Gitlab Practical

- Version control
- Branches and Commits
- Resolving conflicts































Agile

- **Jira**
 - Product by Atlassian
 - Uses both Scrum and Kanban methodologies
 - Interactive task management tool
- **Scrum**
 - Workflow is time-boxed (Sprints)
 - Specific roles
 - Designed to regulate workflow and ensure deployment efficiency
- **Kanban**
 - Not regulated work-time
 - No specific roles
 - Flexibility and adapting to capacity to keep the workflow moving




All are visual tools to help productivity and task optimization

Day Two, End of Daily Scrum



PBI In Sprint	To Do	Doing	Done
	 	 	
	 	 	
	 	 	
	  		
	  	 	
			
			

Legend

-  Analysis
-  Coding
-  Testing

Sprint Goal
Enable User
Capability Z



Teams in Space
Software project



Backlog



Board



Reports



Releases



Components



Issues



Repository



Add item



Settings



Board



Quick filters ▾

Assignee ▾

Sprint ▾

TO DO 5

IN PROGRESS 5

IN REVIEW 2

DONE 8

▽ Super important 11 issues

When requesting user details the service should return prior trip info

SEESPACEEZ PLUS



TIS-37

Engage Jupiter Express for outer solar system travel

SPACE TRAVEL PARTNERS



5

TIS-25



Create 90 day plans for all departments in the Mars Office

LOCAL MARS OFFICE



9

TIS-12

Requesting available flights is now taking > 5 seconds

SEESPACEEZ PLUS



3

TIS-8



Engage Saturn Shuttle Lines for group tours

SPACE TRAVEL PARTNERS



4

TIS-15



Establish a catering vendor to provide meal service

LOCAL MARS OFFICE



4

TIS-15



Engage Saturn Shuttle Lines

Register with the Mars Ministry of Revenue

LOCAL MARS OFFICE



3

TIS-11

Draft network plan for Mars Office

LOCAL MARS OFFICE



3

TIS-15



Add pointer to main css file to instruct users to create child themes

LARGE TEAM SUPPORT



TIS-56

Homepage footer uses an inline style - should use a class

LARGE TEAM SUPPORT



5

TIS-68



Engage JetShuttle SpaceWays for travel

SPACE TRAVEL PARTNERS



5

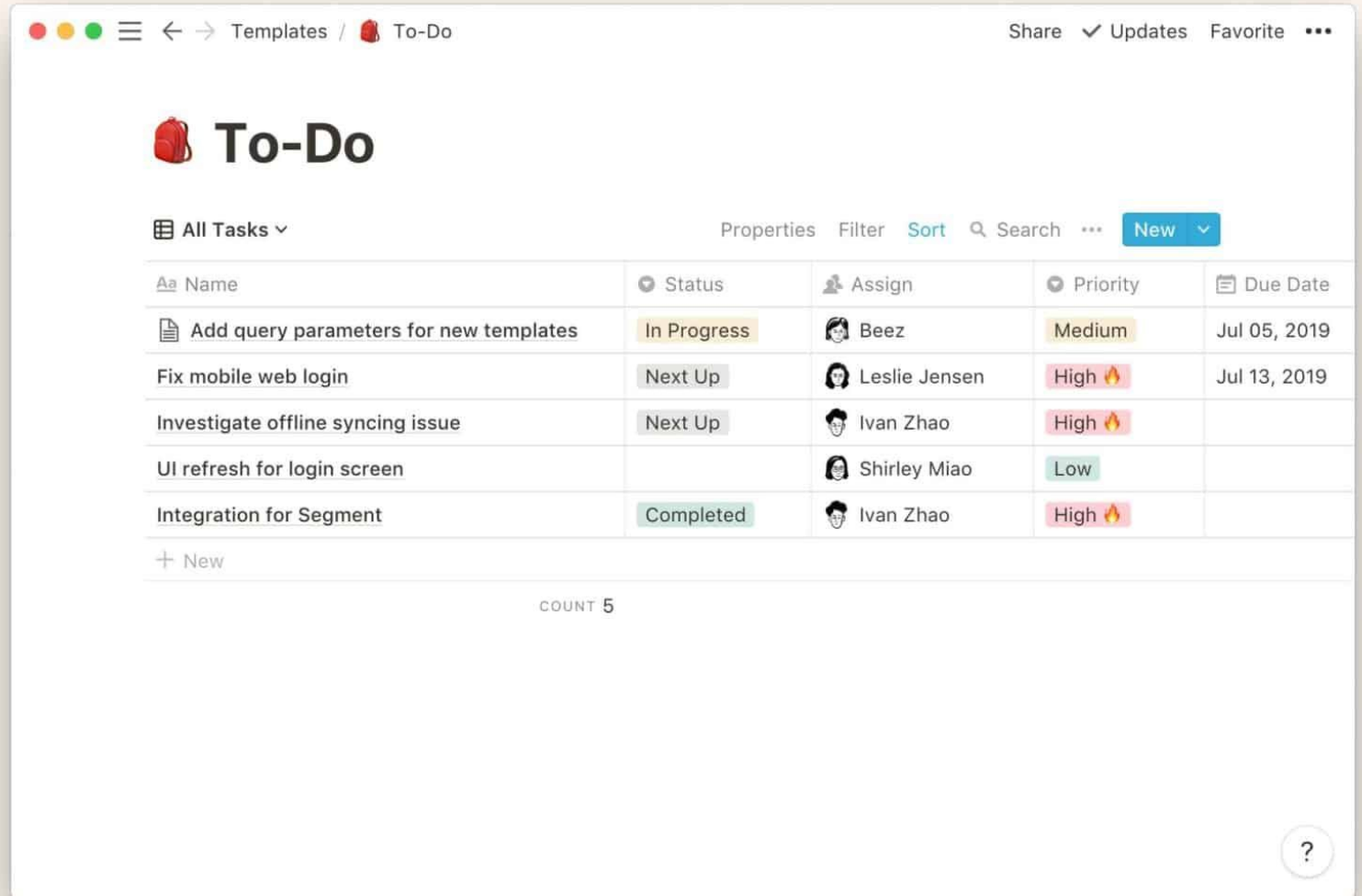
TIS-23



Release



Notion - All in one



The screenshot shows a Notion database titled "To-Do" with a red backpack icon. The interface includes a header bar with window controls, navigation links, and action buttons like "Share", "Updates", and "Favorite". Below the title, there's a section for "All Tasks" with options for "Properties", "Filter", "Sort", "Search", and a "New" button. The database table has five columns: "Name", "Status", "Assign", "Priority", and "Due Date". It contains five task entries with various statuses and priorities. At the bottom, there's a "+ New" button and a "COUNT 5" indicator. A help icon (?) is located in the bottom right corner.

To-Do				
All Tasks ▾				
Name	Status	Assign	Priority	Due Date
Add query parameters for new templates	In Progress	Beez	Medium	Jul 05, 2019
Fix mobile web login	Next Up	Leslie Jensen	High 🔥	Jul 13, 2019
Investigate offline syncing issue	Next Up	Ivan Zhao	High 🔥	
UI refresh for login screen		Shirley Miao	Low	
Integration for Segment	Completed	Ivan Zhao	High 🔥	
+ New				
COUNT 5				

Project Structure

Cookiecutter

- Automatic creation of project structure
- Save templates
- Recreate as You go

Tests - Why do we NEED them?

1. Developer writes code
2. Git add, git commit, git push, merged to main
3. Maintainer/Reviewer is on vacation
4. Code goes to production
5. Code doesn't work
6. App breaks
7. No profit
8. Maintainer/Reviewer disturbed on a faraway island in the middle of the pacific

Tests - Why do we NEED them?

1. Developer writes some code
2. Git add, git commit, git push - Test goes red!
3. Merge reverted!
4. App still works!
5. Profit!
6. Maintainer/Reviewer happily enjoys his vacation

Tests - Practical Examples from the industry

- Quality Assurance and Bug Fixing
- Testers and Developers are two separate teams writing completely different code!
- Developers don't like testers as their job is to break the code of the devs
- Testers think that developers are lazy bums

Tests - Practical Examples from the industry

Software tester goes into a bar.

Orders a beer.

Orders 0 beers.

Orders 999999999 beers.

Orders a bear.

Orders -1 beers.

Orders hdtseatfibkd.

First real customer walks into a bar and asks where the bathroom is.

The bar bursts into flames killing everyone inside.

Tests - Practical Examples from the industry

All companies are testing.

But some can afford not to test in production!

It a little bit like early-access games

- Pay us before official release
- So we don't have to pay testers
- ... profit?



Tests - Practical Examples from the industry

Tested mindset:

- Finding holes that were omitted in development
- When a test fails - quickly point developer to the cause
- Making sure the test coverage is as complete as possible
- Foolproof the app
- Create usage scenarios that reveal weaknesses
- Keep the application working in development and after release
- Make sure the app will pass certifications

Tests - Practical Examples from the industry

Tests to Pass

- Plan the functionality and define requirements
- Project indicators are clearly defined before the project starts
- Write the test structure with the final functionality in mind
- Developer add functionality until all tests are passed

Tests to Fail

- Code tested already exists and/or is on a completely different platform:
 - E.g Wireless communication with other devices
- Certain functionalities need to pass standards (for certification)
- Catching bugs

Tests to Fail Practical - Meter Mystery!

- You are given a water meter that communicates its status wirelessly
- Your job is to check if the meter will pass certification
- You are given requirements for the certification
- Write any tests You deem necessary to check if meter could be certified
- Report any discrepancies from the requirements

Write me a private message on Slack if You think You got it all!

TDD – Test Driven Development

- We are writing tests to pass!
- Create the test structure beforehand
- Write code until You pass!

This has some setbacks:

- Rigid test structure often obstructs functionality
- Lack of flexibility
- Stacking tests can result in a mess

Is there an alternative?

- Do not write any tests ;p
 - Better have SOME rigid tests than none at all
- **BDD :o**

BDD - Behaviour Driven Development

- Focus on behavior or interactions of the system
- Human readability
 - Scenarios!
- Given-When-Then
 - **Given:** Preconditions or initial context.
 - **When:** Actions or events.
 - **Then:** Expected outcomes or results.

Technologies:

- Pytest-bdd
- Behave
- Lettuce
- gherkin

BDD – Behaviour Driven Development

- Gherkin

Feature: Addition

Scenario: Add two numbers

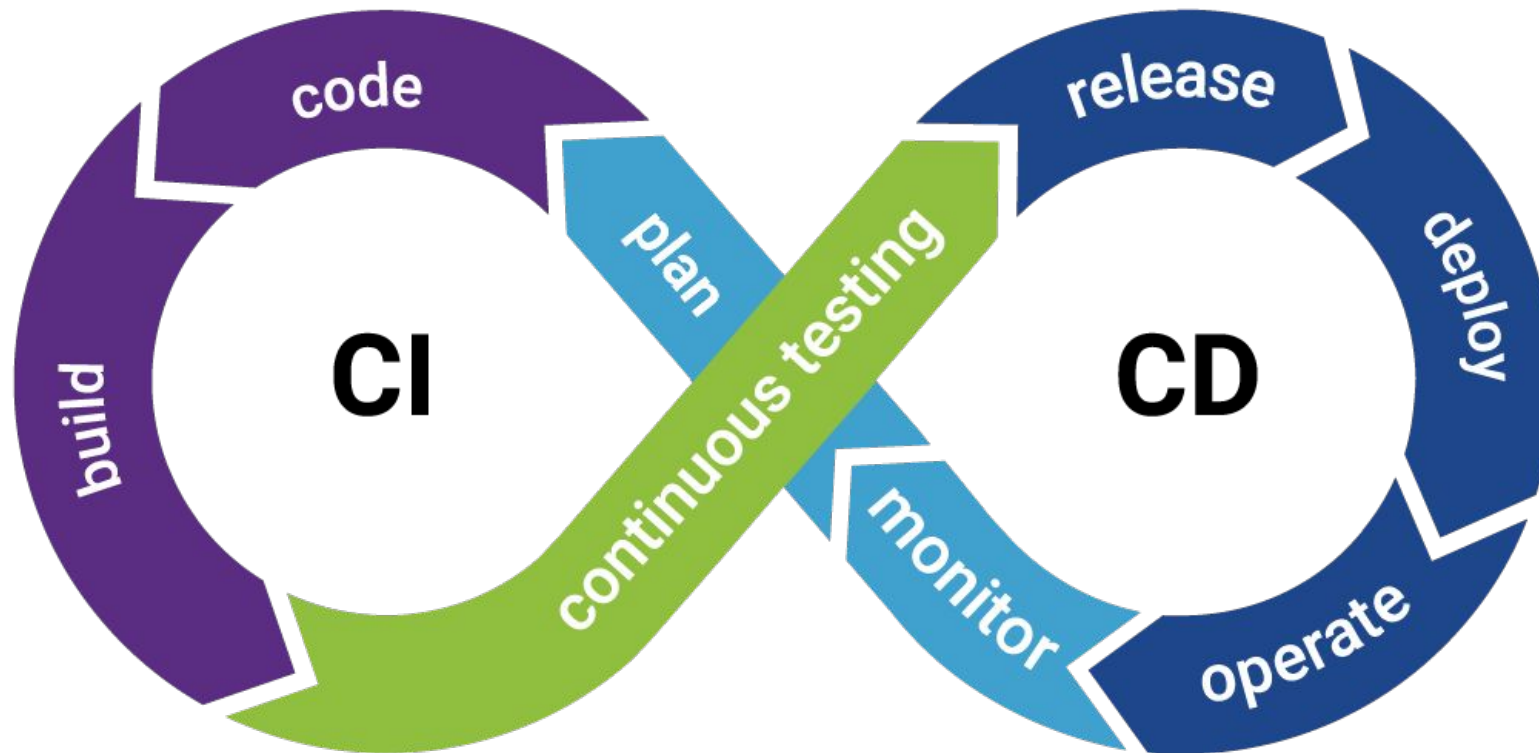
Given I have two numbers 2 and 3

When I add them together

Then I should get the result as 5

CI/CD - DevOps

- Continuous Integration / Continuous Delivery



CI/CD - DevOps

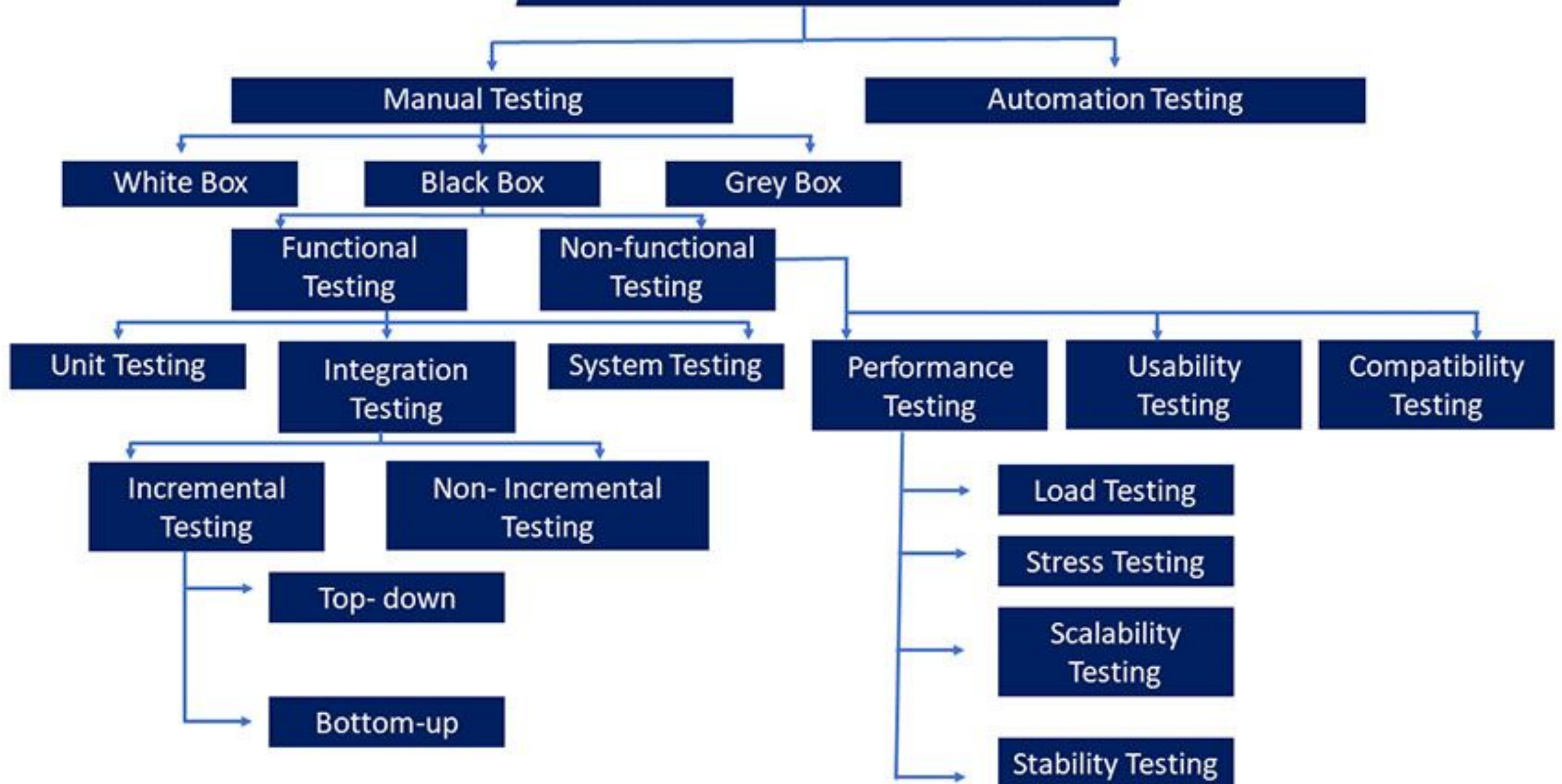
- Funny integrated in GitHub and GitLab
- Automatic scripting applied to branches (usually master or release)
- Script runs stages e.g.:
 - Setup
 - Build
 - Test
 - Deploy
- The scripts execution is done before merging with production

If anything goes bad, merge is reverted!

ISTQB

- International Software Testing Qualification Board
 - Definitions of test types and levels
 - Best practices and practical examples of test coverage
 - Agile test development
 - Moderately priced Certification **(this is not an Ad)**
 - Good way into first python development job is to start in QA (testing)
 - Knowledge of ISTQB is very important for that!

Types of Software Testing



Unittest or pytest?

Well they are different things!

- Unit test is great for unit testing (as the name suggests), but it is not so popular with anything more.
- Unittest is included with any modern python distribution
- Amazing for testing singular modules, objects or functions if they work correctly

Unittest or pytest?

- Pytest is an external library, but become dominantly popular
- It is convenient and flexible
- Can perform unit tests and integration tests at any complexity
- Highly customizable with fixtures
- Compatible with many proprietary repositories (like Allure)

Let's do some co.. **Programming!**

GitLab Project Showcase

- Create a repository on GitLab
 - Use of Cookiecutter repository creation
- Create issues and milestones
- Configure the build with tests

GitLab CI/CD Pipeline - Practical

- Configure CI/CD Pipeline
- Create a branch
- Commit functions and their tests on the branch
- Execute the pipeline
- Create another branch and break some code
- Try to merge the broken code to observe

Sources and Inspirations

- The Zen of Python
 - Tim Peters (2004)
- The Art of Computer Programming
 - Donald E. Knuth (1968 - Still incomplete!)
- Clean Code: A Handbook of Agile Software Craftsmanship
 - Robert C. Martin (2008)
- The Power of Ten – NASA/JPS Coding Guidelines
 - Rules for Developing Safety Critical Code
 - Gerard J. Holzmann (?)

The Zen of Python (PEP 20)

import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

The Zen of Python (PEP 20)

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

The Zen of Python (PEP 20)

There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than **right** now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those

Python – Best Practices

- Comments and Docstrings!
- Explicit Typing
- Variable naming conventions
- No **MAGIC NUMBERS**

Variable naming conventions

#Legal variable names:

myvar = "John" < ok

my_var = "John" < BETTER

_my_var = "John"

myVar = "John" < Classes

MYVAR = "John" < CONSTANTS

myvar2 = "John"

#Illegal variable names:

2myvar = "John"

my-var = "John"

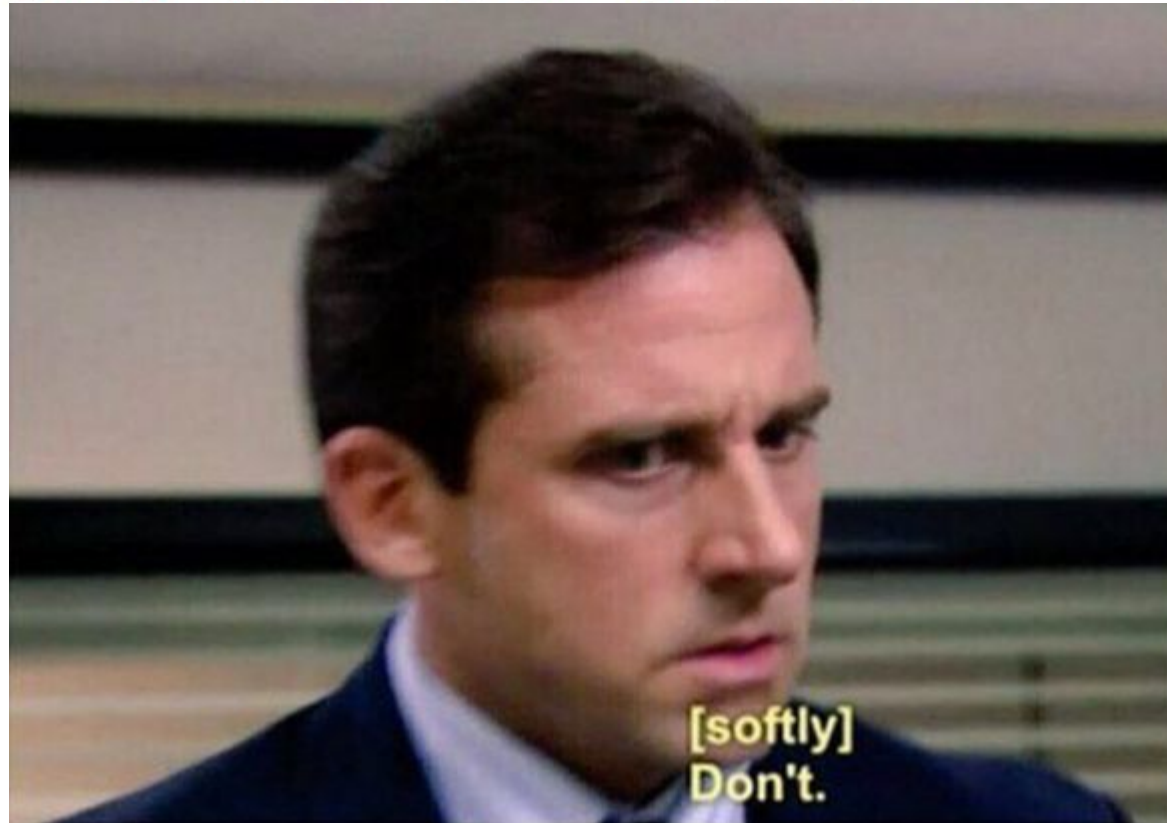
my var = "John"

RESERVED

and	except	lambda	with
as	finally	nonlocal	<u>while</u>
assert	false	None	yield
break	for	not	try
class	from	or	is
continue	global	pass	else
def	if / elif	raise	True
del	import	return	in

Variable naming conventions

`the_best_member_of_the_group_is_what_is_held_in_this_variable`



[softly]
Don't.

Variable naming conventions - others

Functions

- Snake Case

```
def process_data():  
    pass
```

Classes

- Camel Case

Model

Variable naming conventions

- Do not mix conventions
- Focus on clarity and descriptiveness
- Don't use it? Don't name it.

```
for _ in range(1000):  
    print('iteration')
```

```
def function_that_returns_a_lot():  
    return 1, 2, 3  
  
_, _, output = function_that_returns_a_lot()  
print(output)
```

Magic Numbers

```
def calculate_area(radius):  
    return 3.14 * radius ** 2
```

```
result = calculate_area(5)  
print(result) # Output: 78.5
```

```
pi = 3.14
```

```
return pi * radius ** 2
```

```
result = calculate_area(5)  
print(result) # Output: 78.5
```


Magic Numbers

```
euros_count = 1000
dollars_count = euros_count * 1.25 # 1250.0
rubles_count = dollars_count * 60  # 75000.0
```

```
print(rubles_count)
```

```
dollars_per_euro = 1.25
rubles_per_dollar = 60
```

```
euros_count = 1000
dollars_count = euros_count * dollars_per_euro      # 1250.0
rubles_count = dollars_count * rubles_per_dollar    # 75000.0
```

```
print(rubles_count)
```

Explicit Typing

- Type Hints
- Explicit Type

```
def calculate_area(radius: float) -> float:
    """
    Calculate the area of a circle given its radius.

    Parameters:
    radius (float): The radius of the circle.

    Returns:
    float: The area of the circle.
    """
    pi: float = math.pi
    return pi * radius ** 2
```

Explicit Typing - Type Hints

```
from typing import List

def record_telegrams(duration_minutes: int) -> List[str]:
    telegrams = []
    # Your implementation here
```

Comments and Docstrings

[PEP257 - Docstrings](#)

- Specific format
- Informative
- Auto import into documentation

```
def calculate_area(radius):  
    """  
    Calculate the area of a circle given its radius.  
  
    Parameters:  
    radius (float): The radius of the circle.  
  
    Returns:  
    float: The area of the circle.  
    """  
    pi = 3.14  
    return pi * radius ** 2  
  
result = calculate_area(5)  
print(result)  # Output: 78.5
```

Automatic Documentation - Sphinx

Fortunately we don't have to copy all those docstrings and compile documents ourselves!

Practical:

- Sphinx automatic documentation
- Lets follow quick-start guide!

The Art of Computer Programming

The Art of Computer Programming" is a comprehensive multi-volume work by Donald E. Knuth that serves as a foundational text in computer science.

- Commonly known as **TAOCP**
 - Algorithms
 - Data structures
 - Principles of computer programming.

The Art of Computer Programming

- Here are some key points about TAOCP:

1. **Authorship:** Donald Knuth, a renowned computer scientist and Stanford University professor, wrote the series. The first volume was originally published in 1968, and additional volumes have been released over the years.
2. **Algorithmic Concepts:** TAOCP delves deeply into the fundamental concepts of algorithms and their analysis. It explores various sorting and searching algorithms, as well as topics like combinatorial algorithms and arithmetic algorithms.
3. **Mathematical Rigor:** Knuth emphasizes mathematical rigor in the analysis of algorithms. The series includes mathematical proofs, equations, and discussions aimed at providing a thorough understanding of the algorithms presented.
4. **Typesetting System:** Knuth is also known for creating the typesetting system TeX, and he used it to typeset TAOCP. The books are recognized for their high-quality typesetting and the use of a special mathematical notation called "Literate Programming," where programs are embedded in a narrative text.
5. **Ongoing Work:** TAOCP is a work in progress, and Knuth has continually updated and expanded the series over the years. The series is planned to have seven volumes in total, with the author intending to cover a wide array of topics related to computer programming.
6. **Legacy:** The series has had a significant impact on the field of computer science and programming. It is considered a classic and has influenced generations of computer scientists and programmers.

Clean Code

- Clean Code: A Handbook of Agile Software Craftsmanship" is a book written by Robert C. Martin, also known as "Uncle Bob."

The first book on code I have encountered as a kid :D

Clean Code

- **Meaningful Names:** Use descriptive and meaningful names for variables, functions, and classes to enhance code readability.
- **Functions:** Keep functions short, focused on a single responsibility, and avoid side effects. Functions should do one thing and do it well.
- **Comments:** Write comments only when necessary, and make sure they add value. The goal is to make the code self-explanatory without relying on comments.
- **Formatting:** Maintain consistent and clean formatting to make the code visually appealing and easy to navigate.
- **Objects and Data Structures:** Hide implementation details behind abstractions and use proper data structures. Objects should hide their data and expose operations.

Clean Code

- **Error Handling:** Use exceptions rather than returning error codes, and handle errors at the appropriate level of abstraction.
- **Testing:** Write comprehensive unit tests to ensure the correctness of code. Aim for high test coverage to catch potential issues early.
- **Code Smells and Refactoring:** Learn to identify code smells (indicators of poor design) and apply refactoring techniques to improve code structure without changing its external behavior.
- **Concurrency:** Write code that is safe and easy to understand in concurrent environments.
- **Emergent Design:** Allow the design of the system to emerge based on the evolving understanding of the requirements. Avoid unnecessary complexity.

Clean Code

The overarching theme is that clean code is not only about aesthetics!

Creating code that is:

- Easy to work with
- Easy to understand
- Easy to modify

The book emphasizes the importance of craftsmanship in software development and provides practical advice on how to achieve it in the context of agile development practices.

The Power of Ten – NASA/JPL Coding Guidelines

1. *Restrict all code to very simple control flow constructs – do not use goto statements, setjmp or longjmp constructs, and direct or indirect recursion.*
2. *All loops must have a fixed upper-bound. It must be trivially possible for a checking tool to prove statically that a preset upper-bound on the number of iterations of a loop cannot be exceeded. If the loop-bound cannot be proven statically, the rule is considered violated.*
3. *Do not use dynamic memory allocation after initialization.*
4. *No function should be longer than what can be printed on a single sheet of paper in a standard reference format with one line per statement and one line per declaration. Typically, this means no more than about 60 lines of code per function.*
5. *The assertion density of the code should average to a minimum of two assertions per function. Assertions are used to check for anomalous conditions that should never happen in real-life executions. Assertions must always be side-effect free and should be defined as Boolean tests. When an assertion fails, an explicit recovery action must be taken, e.g., by returning an error condition to the caller of the function that executes the failing assertion. Any assertion for which a static checking tool can prove that it can never fail or never hold violates this rule (i.e., it is not possible to satisfy the rule by adding unhelpful “assert(true)” statements).*
6. *Data objects must be declared at the smallest possible level of scope.*
7. *The return value of non-void functions must be checked by each calling function, and the validity of parameters must be checked inside each function.*
8. *The use of the preprocessor must be limited to the inclusion of header files and simple macro definitions. Token pasting, variable argument lists (ellipses), and recursive macro calls are not allowed. All macros must expand into complete syntactic units. The use of conditional compilation directives is often also dubious, but cannot always be avoided. This means that there should rarely be justification for more than one or two conditional compilation directives even in large software development efforts, beyond the standard boilerplate that avoids multiple inclusion of the same header file. Each such use should be flagged by a tool-based checker and justified in the code.*
9. *The use of pointers should be restricted. Specifically, no more than one level of dereferencing is allowed. Pointer dereference operations may not be hidden in macro definitions or inside typedef declarations. Function pointers are not permitted.*
10. *All code must be compiled, from the first day of development, with all compiler warnings enabled at the compiler’s most pedantic setting. All code must compile with these setting without any warnings. All code must be checked daily with at least one, but preferably more than one, state-of-the-art static source code analyzer and should pass the analyses with zero warnings.*

The Power of Ten – NASA/JPL Coding Guidelines

1. Limit Line Length: Keep lines of code relatively short for readability.

Bad example

```
long_variable_name = calculate_some_really_long_function_name  
(argument1, argument2, argument3, argument4)
```

Good example

```
result = calculate_short_function(arg1, arg2)
```

The Power of Ten – NASA/JPL Coding Guidelines

2. Limit Function Length: Keep functions short and focused.

Bad example

```
def complex_function(arg1, arg2, arg3):  
    # 50 lines of code here  
    ...
```

Good example

```
def simple_function(arg1, arg2):  
    # A few lines of code here  
    ...
```

The Power of Ten – NASA/JPL Coding Guidelines

3. Limit File Length: Keep files reasonably small and focused.

Bad example:

LargeFile.py

Good example:

SmallFile1.py, SmallFile2.py, etc.

The Power of Ten – NASA/JPL Coding Guidelines

4. Limit Routine Length: Similar to function length, keep routines (methods) short.

Bad example

```
class SomeClass:  
    def long_routine(self, arg1, arg2):  
        # 50 lines of code here  
        ...
```

Good example

```
class SomeClass:  
    def short_routine(self, arg1, arg2):  
        # A few lines of code here  
        ...
```


The Power of Ten – NASA/JPL Coding Guidelines

5. Limit Variable Scope: Minimize the scope of variables.

Bad example

```
def some_function():  
    global some_var  
    some_var = 10  
    ...
```

Good example

```
def some_function():  
    some_var = 10  
    ...
```

The Power of Ten – NASA/JPL Coding Guidelines

6. Use Constants for Magic Numbers: Avoid magic numbers; use named constants.

Bad example

```
if x > 86400:
```

```
    ...
```

Good example

```
SECONDS_IN_DAY = 86400
```

```
if x > SECONDS_IN_DAY:
```

```
    ...
```

The Power of Ten – NASA/JPL Coding Guidelines

7. Limit Data Hiding: Don't use unnecessary access specifiers in classes.

Bad example

```
class SomeClass:
    def __init__(self):
        self.__private_var = 10
    def get_private_var(self):
        return self.__private_var
```

Good example

```
class SomeClass:
    def __init__(self):
        self.private_var = 10
```

The Power of Ten – NASA/JPL Coding Guidelines

8. Avoid Using Macros: In Python, this translates to avoiding global constants or variables.

Bad example

```
PI = 3.14159
```

Good example

```
import math  
pi = math.pi
```

The Power of Ten – NASA/JPL Coding Guidelines

9. Avoid Compound Statements: Keep statements simple and separate.

Bad example

```
if condition1 and condition2:
```

```
    ...
```

Good example

```
if condition1:
```

```
    if condition2:
```

```
        ...
```

The Power of Ten – NASA/JPL Coding Guidelines

10. Avoid Negative Logic: Write conditions in positive form when possible.

Bad example

if not is_invalid:

...

Good example


if is_valid:

...

General Rules

- First code for correctness – then for clarity
- Premature optimization is the root of all evil
- Comment the code! While Coding! (You will thank Yourself later)
- Readability becomes a priority AFTER the code works

- Code optimization

- 
1. Code works
 2. Clean the code
 3. Optimize running efficiency

- Make sure somebody else can use what You did!

Important Developer Acronyms

- PEP
 - TDD/BDD
 - CI/CD
 - ISTQB
 - SOLID
-
- DRY – Don't Repeat Yourself
 - KISS – Keep It Stupid Simple
 - YAGNI – You Ain't Gonna Need It
 - WYSIWYG – What You See is What You Get

Personal Favourites:

- PEBCAK – Problem Exists Between Chair and Keyboard
- PICNIC – Problem In Chair Not In Computer

S.O.L.I.D.

- **S**.ingle Responsibility
 - One class – one responsibility
- **O**.pen-Closed
 - Open for extension, Closed for Modification
- **L**.iskov Substitution
 - Replace parent with child – nothing breaks
- **I**.terface segregation
 - Do not depend on something you do not explicitly use
- **D**.ependency Inversion
 - Interfaces between classes and objects

Single Responsibility

- One class – one responsibility

```
class UserManager:
    new *
    def __init__(self):
        self.db = Database()

    new *
    def add_user(self, user):
        self.db.insert(user)

    new *
    def remove_user(self, user_id):
        self.db.delete(user_id)
```

```
class Database:
    1 usage new *
    def insert(self, user):
        # Logic to insert user into database

    1 usage new *
    def delete(self, user_id):
        # Logic to delete user from database
```

O.pen-Closed

- Open for extension, Closed for Modification

```
class Shape:
    new *
    def area(self):
        pass

new *
class Rectangle(Shape):
    new *
    def __init__(self, width, height):
        self.width = width
        self.height = height
    new *
    def area(self):
        return self.width * self.height
```

```
class Circle(Shape):
    new *
    def __init__(self, radius):
        self.radius = radius
    new *
    def area(self):
        return 3.14 * self.radius * self.radius
```

Liskov Substitution

- Replace parent with child – nothing breaks

```
class Bird:
    new *
    def fly(self):
        pass

new *
class Duck(Bird):
    new *
    def fly(self):
        print("Duck flying")
```

```
class Ostrich(Bird):
    new *
    def fly(self):
        raise NotImplementedError("Ostrich cannot fly")
```

Interface segregation

- Do not depend on something you do not explicitly use

```
class Printer:
    new *
    def print(self, document):
        pass
```

```
2 usages new *
class Scanner:
    new *
    def scan(self, document):
        pass
```

```
new *
class Photocopier(Printer, Scanner):
    pass
```

```
class MultiFunctionDevice(Printer, Scanner):
```

```
    new *
    def print(self, document):
        # Logic to print document
```

```
    new *
    def scan(self, document):
        # Logic to scan document
```

Dependency Inversion

- Interfaces between classes and objects

```
class Database:
    1 usage (1 dynamic)    new *
    def get_data(self):
        pass
```

```
class DataManager:
    new *
    def __init__(self, database):
        self.database = database
    new *
    def process_data(self):
        data = self.database.get_data()
        # Process data

db = Database()
data_manager = DataManager(db)
```



PEP – Python Enhancement Protocol

PEP8 - [Style Guide for Python Code](#)

PEP 20: The Zen of Python - A collection of guiding principles for writing computer programs in Python.

PEP 257: Docstring Conventions - Defines conventions for writing docstrings (documentation strings) for Python modules, functions, classes, and methods.

PEP 484: Type Hints - Introduces a syntax for annotating the types of variables, function arguments, and return values in Python code.

PEP 333: Python Web Server Gateway Interface (WSGI) - Defines a standard interface between web servers and Python web applications or frameworks.

PEP 3333: Python Web Server Gateway Interface v1.0.1 - An update to PEP 333 to clarify certain aspects and improve compatibility.

PEP 572: Assignment Expressions - Introduces the "walrus operator" (`:=`), allowing assignment expressions within other expressions.

PEP 498: Literal String Interpolation - Introduces f-strings, a more concise and readable way to format strings in Python.

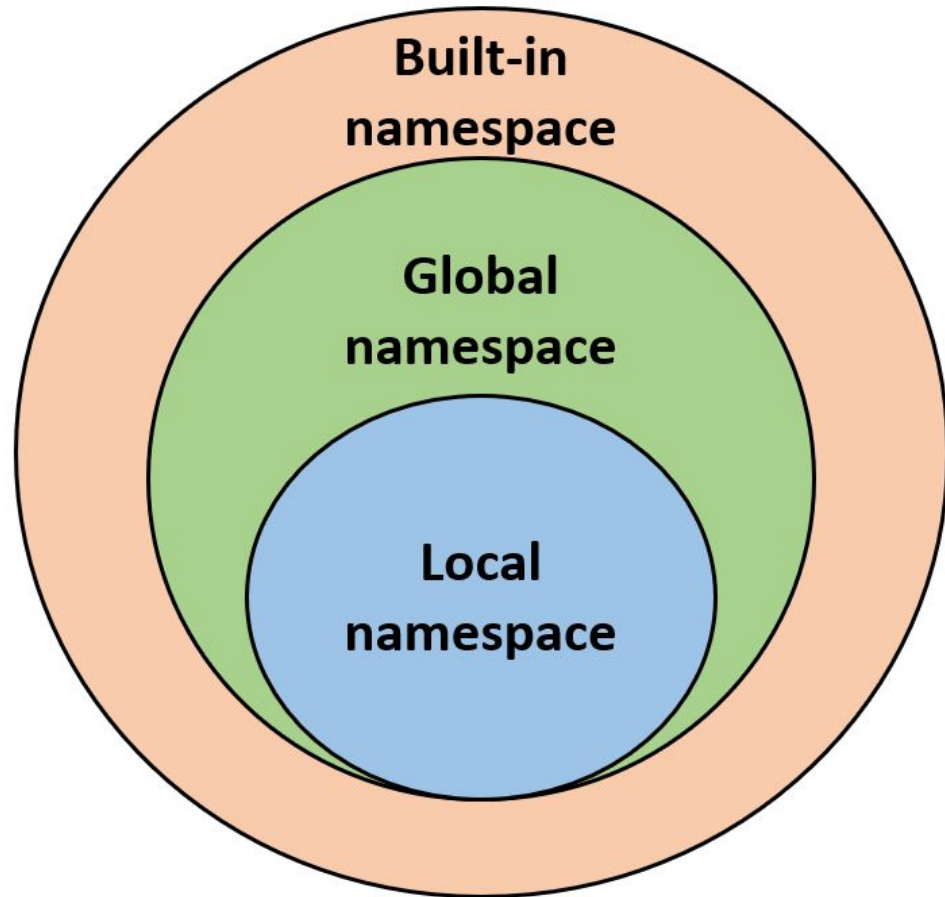
PEP 3107: Function Annotations - Introduces a syntax for adding metadata (annotations) to function parameters and return values.

PEP 394: The "python" Command on Unix-like Systems - Defines the expectations for the python command and its variants on Unix-like systems, particularly in relation to Python 2 and Python 3 coexistence.

Pre-commit and style check

- Pre-Commit
 - Requires configuration file (provided)
- Flake8
 - Can be installed into PyCharm as external tool
- Black
 - Executed from console, repairs all code in repo

Python Scopes



Type of Namespaces

```
namespace.py
1 x = 10
2 y = 20
3
4
5
6
def outer():
    z = 30
    def inner():
        x = 30
        print(f'x is {x}')
        print(f'z is {z}')
        print(f'y is {y}')
        print(len("abc"))
    inner()
outer()

outer() > inner()
namespace
/Users/pankaj/Documents/PycharmProjects/PythonTutorialPro/venv/bin/p
x is 30
z is 30
y is 20
3
Process finished with exit code 0
```

Annotations in the code editor:

- global**: Points to `x = 10` and `y = 20` in the global scope.
- enclosed**: Points to `z = 30` in the `outer()` function scope.
- local**: Points to `x = 30` in the `inner()` function scope.
- built-in namespace has len() function**: Points to the `len()` function call in the `inner()` function.

Thank You for Your attention!