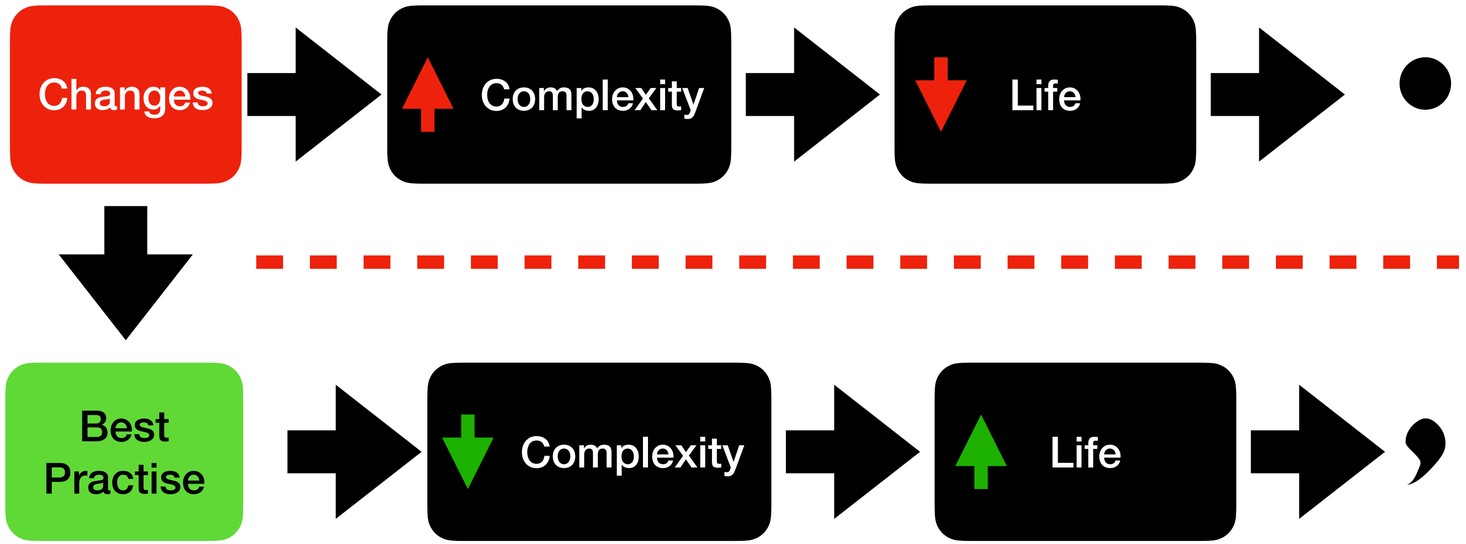
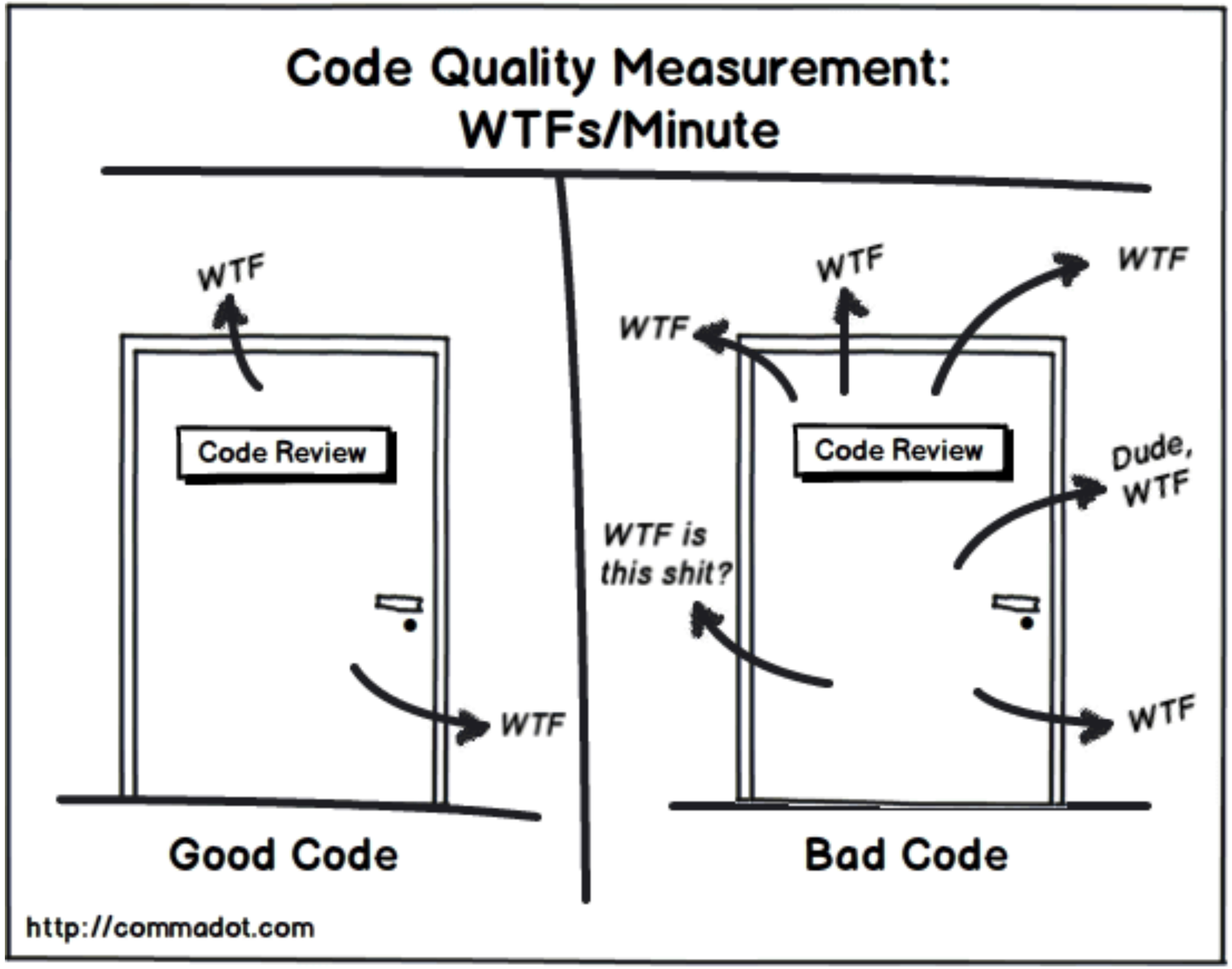
Coding Style

My Message – Working Code isn't Enough



Goal #2 - Clean Code - Coding Style



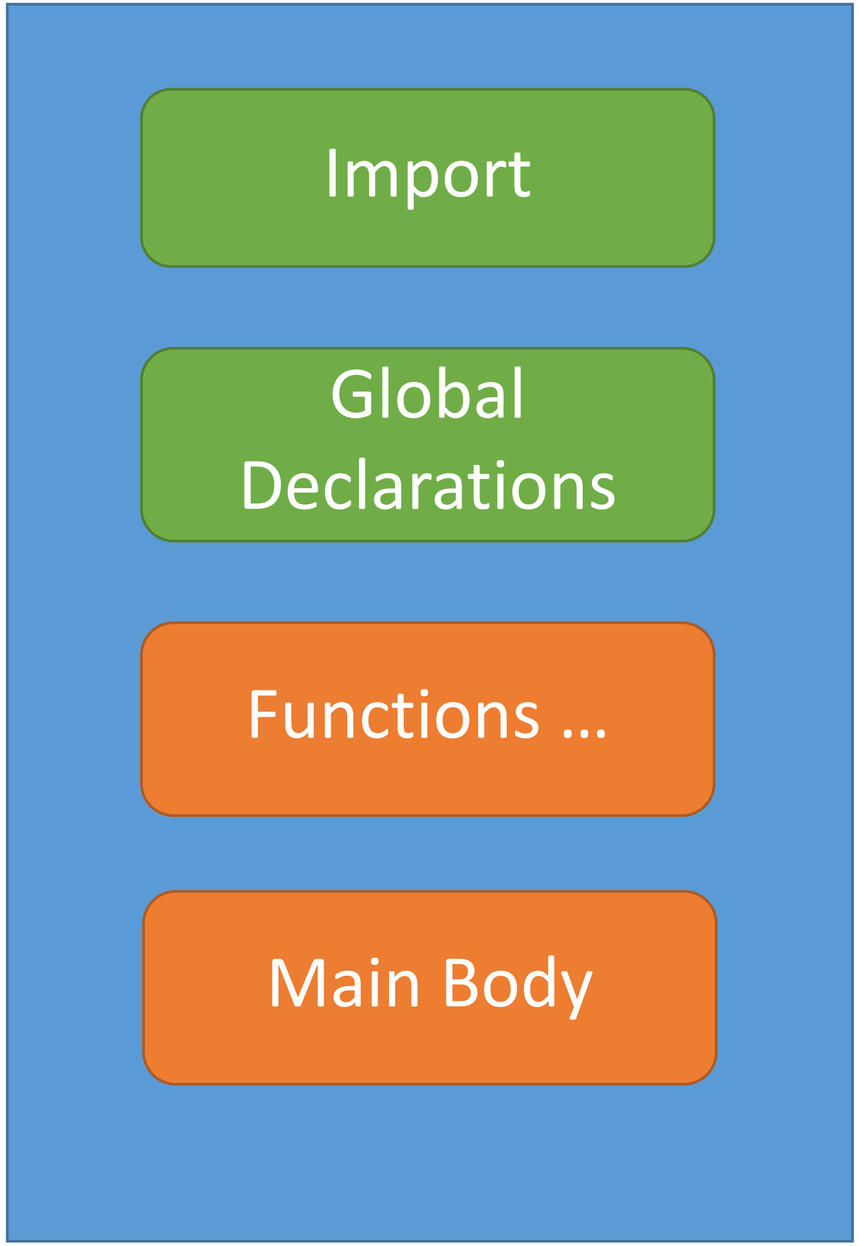
coding style

- program layout (import, declarations, functions, main body)
- Indentation – functions (parameters), python objects (lists, dictionary)
- whitespaces – statements (assignment, comparisons, boolean)
- blank lines
- comment (up-to-date, indentation, inline)
- max. line length
- compound statements (split them)
- naming convention - snake_case

Coding Style - Others

- Too many arguments
- Too many local variables
- Too many boolean expressions in if statement
- Too many branches
- Too many statements (per function)
- Statement line too long
- Unused argument
- Unused variable (global, local, function)

layout



PEP 8 – Coding Styles

- Code lay-out
  - Indentation
  - Tabs or Spaces?
  - Maximum Line Length
  - Should a line break before or after
  - Blank Lines
  - Source File Encoding
  - Imports
  - Module level dunder names
- Whitespace in Expressions and Statements
  - Pet Peeves
  - Other Recommendations
- Comments
  - Block Comments
  - Inline Comments
  - Documentation Strings
- Naming Conventions
  - Overriding Principle
  - Descriptive: Naming Styles
  - Prescriptive: Naming Conventions

indentation, comments, default parameters

```
5   #
6   # A function to show indentation for parameters
7   #
8   def a_function_with_many_parameters( p_user_name, p_user_mailing_address, p_user_home_address, p_user_conta
9       print('hello world')
10
11  def a_function_with_many_parameters(
12      p_user_name,
13      p_user_mailing_address,
14      p_user_home_address,
15      p_user_contact_number,
16      p_user_receive_email = 1)
17      print('hello world')
18
19  def a_function_with_many_parameters(
20          p_user_name,                # first name only
21          p_user_mailing_address,     # full email address
22          p_user_home_address,        # street, province, postal code
23          p_user_contact_number       # (area code)xxx-xxxx
24          p_user_receive_email=1      # receive email flag
25          )
26      print('hello world')
```

whitespaces

assignment

boolean

parameters

```
def whitespace_in_statement( x , y , z):
    if (x+y>z)and y!=z :
        ret= ( x[1: 4] * 2 )+( y / z )

def whitespace_in_statement(x, y, z):
    if (x+y > z) and y != z:
        ret = (x[1:4]*2) + (y/z)
```

colon

slicing

Self-documenting

- Use spacing & parentheses to be readable and clarify precedence.
- Use consistent ordering of the terms in subexpressions.
- Use subexpressions and extra variables instead of long complicated expressions.

Example

1. For each row in the table below, write X or Y in the last column to indicate which option is better. *Sample answers are shown in blue italics.*

Option X	Option Y	X/Y
`s1=i1*c1+i2*c2;`	`s1 = i1*c1 + i2*c2;`	*Y*
`s1=(i1*c1)+(i2*c2);`	`s1=i1*c1+i2*c2;`	*X*
`s1 = c1*i1 + i2*c2;`	`s1 = i1*c1 + i2*c2;`	*Y*

`total = nCD*sCD +`
`(nCD*cCD + nMP3*cMP3) *`
`(1+rateTax);`
	`cost = nCD *cCD +`
	`nMP3*cMP3;`
	`ship = nCD*sCD;`
	`tax = cost * rateTax;`
	`total = cost + tax + ship;`
		Y

compound statements

```
if check_something(): call_funct_a(); call_another_funct()
call_funct_b(); call_funct_c(); call_funct_d()

if check_something():
    call_funct_a()
    call_another_funct()
call_funct_b()
call_funct_c()
call_funct_d()
```

Naming Convention - snake_case

- “snake_case” is the naming convention in which each space is replaced with an underscore (_) character, and words are written in lowercase.
  - Ex: pick_a_number(), get_correct_cnt(), get_misplaced_cnt()
- Must use snake_case style for all functions, variables (except constant), parameters
- Scope
  - global vs local variables
  - constants
  - function parameters
  - function names
  - import names

snake_case - examples

- local variable
  - cnt_student (optional: _cnt_student)
- global variables
  - g_cnt_student
- global constants
  - MAX_GUESS_ATTEMPT
- functions
  - get_student_count()
- function parameters
  - upper_limit (optional: p_upper_limit)

Comments - Excuses vs Reasons

- Excuses
  - Well written code should be self-documenting
  - Don’t have time
  - Comments are useless and out-of-date - Except for major-scope change
- Purposes
  - Describe things that aren’t obvious; describe overall behaviors, including input(s), output(s), prerequisites, plus usage examples.
- Useful information
  - High Level (overall abstraction, thoughts, ideas, reasoning, structure)
  - Low Level (add precise information about specific logic)
  - Same Level (likely to repeat the code itself)

Comment - Be Useful

- Don’t repeat code;
  - szSnakeBody = 4 # Size of the snake body (not useful)
  - szSnakeBody = 4 # Total length of the snake in multiple of the head size
  - szPadding = 4 # Padding space (not useful)
  - szPadding = 4 # The margins on both sides (left & right) in pixels

```
/*
 * The horizontal padding of each line in the text.
 */
private static final int textHorizontalPadding = 4;
```

```
/*
 * The amount of blank space to leave on the left and
 * right sides of each line of text, in pixels.
 */
private static final int textHorizontalPadding = 4;
```

Commenting - Function & Module

- Module (High Level)
  - Overall abstraction, design, process flow, major components, assumptions.
- Function
  - An opening sentence or two describing the general usage of the function.
  - Include description of each parameter and the return (output), including the type, any constraints, any dependencies between parameters as well as the type of the return value.
  - Include examples.

Docstring - Module-Level, Function-Level

```
'''
Your module's verbose yet thorough docstring here.
'''
import os

def add_binary(a:int, b:int) -> str:
    '''
    Parameters:
        a (int): A decimal integer
        b (int): Another decimal integer

    Returns:
        Returns the sum of two decimal numbers in binary digits.
    '''
    binary_sum = bin(a+b)[2:]
    return binary_sum
```

Clean Code - Examples

Meaningful Names

```python
# A dictionary of families who live in each city
mydict = {
    "Midtown": ["Powell", "Brantley", "Young"],
    "Norcross": ["Montgomery"],
    "Ackworth": []
}

def a(dict):
    # For each city
    for p in dict:
        # If there are no families in the city
        if not mydict[p]:
            # Say that there are no families
            print("None.")
```

```python
families_by_city = {
    "Midtown": ["Powell", "Brantley", "Young"],
    "Norcross": ["Montgomery"],
    "Ackworth": [],
}

def no_families(cities):
    for city in cities:
        if not families_by_city[city]:
            print(f"No families in {city}.")
```

nested branches

```
def correct_fruits(fruits):
    if len(fruits) > 1:  # [too-many-nested-blocks]
        if "apple" in fruits:
            if "orange" in fruits:
                count = fruits["orange"]
                if count % 2:
                    if "kiwi" in fruits:
                        if count == 2:
                            return True
    return False
```

```
def correct_fruits(fruits):
    if len(fruits) > 1 and "apple" in fruits and "orange" in fruits:
        count = fruits["orange"]
        if count % 2 and "kiwi" in fruits and count == 2:
            return True
    return False
```

pick_a_number(length)

```
s = ''
while len(s) < length:
    r = random.randint(0, 9)
    if str(r) not in s:
        s += str(r)
return s
```

```
digits = list(range(10))
random.shuffle(digits)
return ''.join(str(d) for d in digits[:length])
```

```
ret = random.sample(string.digits, length)
return ''.join(ret)
```

get_correct_count(secret, gues)

```
correct = 0
for i in range(len(secret_number)):
    if secret_number[i] == guess_number[i]:
        correct += 1
return correct
```

```
correct = [s for s,g in zip(secret, guess) if s == g]
return len(correct)
```

Multiple Assignment, ternary if

- Swapping values of two variables (a,b):

```
tmp = a
a = b
b = a
```

```
a, b = b, a
```

- Returning a value based on a single condition (not nested):

```
If condition-A:
    return value-A
else:
    return value-B
```

```
return value-A if condition-A else value-B
```

- Using list comprehension:

```
# double each value
for idx in range(len(listA)):
        listA[idx] = listA[idx] * 2
```

```
[ x * 2 for x in listA ]
```

- Using Map (Reduce, Partial)

```
ans = 0
for idx in range(len(listA)):
        ans += foo(listA[idx])
return ans
```

```
return sum( map(foo, listA) )
```

- Using dictionary as lookup

```
if ans = 'A':
        fooA()
elif ans = 'B':
        fooB()
elif ans = 'C':
        fooC()
elif ans = 'D':
        fooD()
```

```
callFoo = { 'A': fooA, 'B': fooB, 'C': fooC, 'D': food }

callFoo[ans]
```