#22 - Coding - Tips

08/06/2025

IF YOU ARE SERIOUS ABOUT PROGRAMMING AND YOU WANT TO DO IT FOR YEARS AND MAYBE DECADES FROM TODAY, THEN THE HOURS YOU PUT INTO DEBUGGING LITTLE THINGS ARE GOING TO HELP YOU RETAIN KNOWLEDGE FOR THE LONG HAUL.

Naming Things

- No need to save on characters!
- Choose descriptive names.
- variable names should describe the content of the variable.
  e.g., if trying to capture a user response to determine whether the user wants to play a game again. Could name the variable as play_again.

Naming Conventions

- Idiomatic names are naming conventions that follow the Python Style Guide.

Idiomatic names →

Category	Name	Notes
Non-constant variables and objects properties	employee	
	number	
	fizz_buzz	
	speed_of_light	
	destination_url	url is an acronym
	_var	meant for internal use
	var_	used to avoid conflict with keywords
Classes	Cat	
	BoxTurtle	
	FlightlessBird	

08/06/2025

Category	Name	Notes
Functions and Methods	parse_url	url is an acronym
	go_faster	
Constant names	DEFAULT_TIMEOUT	SCREAMING_SNAKE_CASE
	MIN_AGE	

Valid but non-idiomatic Names

Category	Name	Notes
Universally non-idiomatic	#	Non ASCII characters
	__var__	except as defined by Python
	user_Profile	should be all lowercase
	userProfile	camelCase is not idiomatic
	milesperhour	Undifferentiated words
Non-constant variables and dictionary keys	FIZZ_BUZZ	SCREAMING_SNAKE_CASE
	fizzBuzz	camelCase (sorta); non-idiomatic
Classes.	cat	Begins with lowercase letter
	make_turtle	snake_case

Invalid Names

Name	Notes
42ndStreet	Begins with a digit
fizz buzz	contains space
fizz-buzz	contains special character except underscore
for	Python keyword

08/06/2025

Avoid Magic Numbers

A magic number is a number or simple value that appears in a program without any information that describes what that number represents.

Use constants to avoid magic numbers:

```
NUMBER_CARDS_IN_HAND = 5

def deal_hand():
    hand = []
    for card_number in range(NUMBER_CARDS_IN_HAND):
        hand.append(deal_card())

    return hand
```

Typically, magic number constants are set at the top level of a program.
Can also define them inside a function.

Make sure the definition of constants is clear:
e.g., when defining the unicode points for the characters `a` and `z`

```
FIRST_CHARACTER_CODE = 97
LAST_CHARACTER_CODE = 122
```

NOT SO CLEAR

```
FIRST_CHARACTER_CODE = ord('a')
LAST_CHARACTER_CODE = ord('z')
```

VERY CLEAR

`ord('a')` returns unicode point of `a`
`ord('z')` returning unicode point of `z`

08/06/2025

Formatting

- Always use four space characters when indenting. Never use tab characters

- When indenting code blocks, remain mindful of the line length to ensure the code remains easy to read and understand.

```
def calculate_sum(a, b):
    return a+b
           ↑↑
           Spaces around
           operator
```

```
total = num1+num2    # bad
total = num1 + num2  # good
```

Mutating Constants

- Constants remain unchanged throughout program's execution. Therefore, AVOID mutating constants.
Use SCREAMING_SNAKE_CASE to indicate constants and treat them as read-only values.

Consider the following:

08/06/2025

```
CARDS = [1, 2, 3]

CARDS.append(4)
print(CARDS) # [1, 2, 3, 4]

CARDS[1] = 'changed'
print(CARDS)  # [1, 'changed', 3, 4]

CARDS.pop(0)
print(CARDS) # ['changed, 3, 4]
```

Despite Python allowing for the above to occur, it is not good practice considering the list object is declared as a constant, indicated by the SCREAMING_SNAKE_CASE naming of CARDS.

Function Guidelines

Limit the responsibility of functions to one thing. It should be short, no longer than 10 lines.

Side effects of functions include the following:

1. The function reassigns any non-local variable. Reassigning the variable in the outerscope would be a side effect.

2. The function modifies the value of any data structure passed as an argument, or accessed directly from the outer scope. Mutating an object is an example of a side effect e.g., appending an element to a list.

3. The function reads from or writes to a file, network connection, browser, or the system hardware. Side effects include printing and reading input from the terminal

NB: A side effect is any change a function makes outside of returning a value

08/06/2025

4. The function raises an exception without handling it.

5. The function calls another function that has side effects

The following fuctions have side effects:

```
# side effect: prints output
# returns: None

def display_total(num1, num2)
    print(num1 + num2)

#side effect: mutates the passed in list
# returns the updated list

def add_to_list(target_list, value_to_append):
    target_list.append(value_to_append)
    return target_list
```

An example of a function with no side effects

```
# side effect: none
# returns: a new number

def computer_total(num1, num2):
    return num1 + num2
```

Most functions should return a useful value or a side effect, but not both.

has meaning to the calling code

08/06/2025

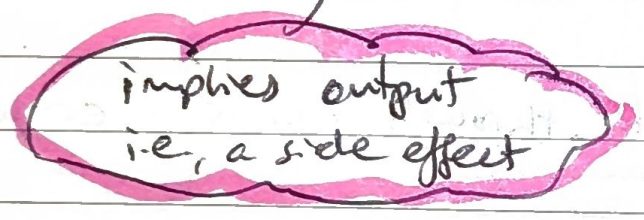
A function that returns an arbitrary value or that always returns the same value (such as `None`) is not returning a useful value

Exceptions to the unwritten rule:

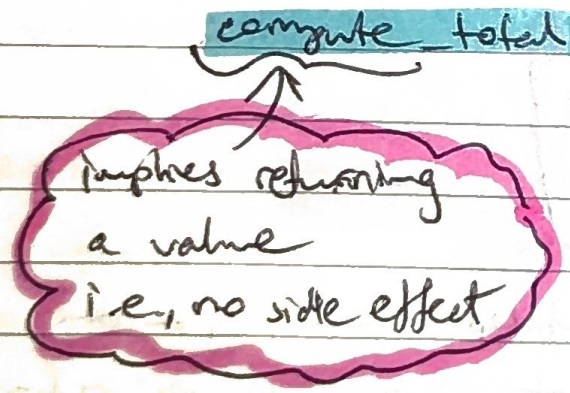
When accessing a database and reading and writing from the terminal are side effects, but may also need to return values.

Function names should be appropriate to the instance of whether a side effect may occur.

e.g., `display_total` for a function that displays a total on a console



`compute_total` for a function that will be returning a value of a computation



09/06/2025

Functions should be at the same level of abstraction

- Functions generally take some input and return some output.
- You should be able to mentally extract a function and work with it in isolation.
- Programming is much simpler if your functions are correctly compartmentalized.

09/06/2025

For example:

deal()
hit()
stay()
} ← This is "what" to do

iterate_through_cards() ← This is "how" to do it.

this one stands out the most.

Best to stick to naming that is in relation to "what" the function does, opposed to "how" it does it.

A better name for iterate_through_cards() would be:

```
draw_card():
    # internally uses iterate_through_cards() or logic.
```

Function Names Should Reflect What They Do

```
def updated_total(total, cards):
    # ... some code here
```

We assume the above function mutates something — one of the arguments or something else. Do not expect it to return anything. If it does, you might have a problem.

Use naming conventions that signify which types of functions mutate vs. which functions return values.

Function Type	Mutates Input	Returns Value	Example function
Mutating	Yes	No	list.append() dict.update()
Returning	No	Yes	sum(), sorted() custom pure functions

09/06/2025.

- Avoid implementing print() and return within the same function.

  "don't print and return"

- Exceptions are the input() function which allows the reading of the user input from the terminal.

- Overall functions should be:
  - like lego blocks to build larger structures
  - not mentally taxing
  - to be organised into classes and objects. (covered later in the curriculum).

Displaying Output

- Some functions only display output.

```
def welcome():
    print("welcome")
```

This function name is not clear on what it does. Best to use a descriptor:

```
print_welcome()
display_welcome()
say_welcome()
```

Miscellaneous Tips

- Always use 4 spaces, not tabs for indentation
- Use names for functions from the perspective of using them later.

  e.g., a function to find the 'ace' within a deck of cards.

  `find_ace():` => `ace = find_ace(cards)`

09/06/2025

- Know when to use a regular **while** loop vs. **while True**

examples:

```
while answer.lower() != 'n':
    print('Continue? (y/n)')
    answer = input()
```

Python will throw an exception of "NameError: name 'answer' is not defined."

Must initialise **answer** before the **while** statement.

①

```
answer = ''
while answer.lower() != 'n':
    print('Continue? (y/n)')
    answer = input()
```

This works, however all the code could be contained in a **while True**.

②

```
while True
    print('Continue? (y/n)')
    answer = input()
    if answer.lower() == 'n'
        break
```

← Instead could remove answer and have:
if input.lower() == 'n'

Feature	Version ①	Version ②
Input before condition	No, needs preset variable	Yes (input is immediate)
Readability	Moderate	Clear
Extensibility	Limited	Easy to add more logic
Initialisation needed?	Yes (answer = '')	No.

while True:

CHATGPT

09/06/2025

A breakdown of how `while True:` works.

```
while True:
    user_input = input("Enter 'n' to quit: ")
    if user_input == 'n':
        break
    print("You entered:", user_input)
```

Step 1 `while True:` starts the loop

As `True` is always true, the loop doesn't stop by itself

Step2 It runs the indented block

```
user_input = input("Enter 'n' to quit: ")
```

The program waits for the user to enter something.

Step3 Check for exit condition

```
if user_input == 'n':
    break
```

If the user enters `"n"`, `break` is run, which immediately exits the loop

Step4 Do something (if not exiting)

```
print("You entered: ", user_input)
```

Step5 Repeat...

Unless the user enters `"n"`, it keeps repeating - asking, checking, printing.