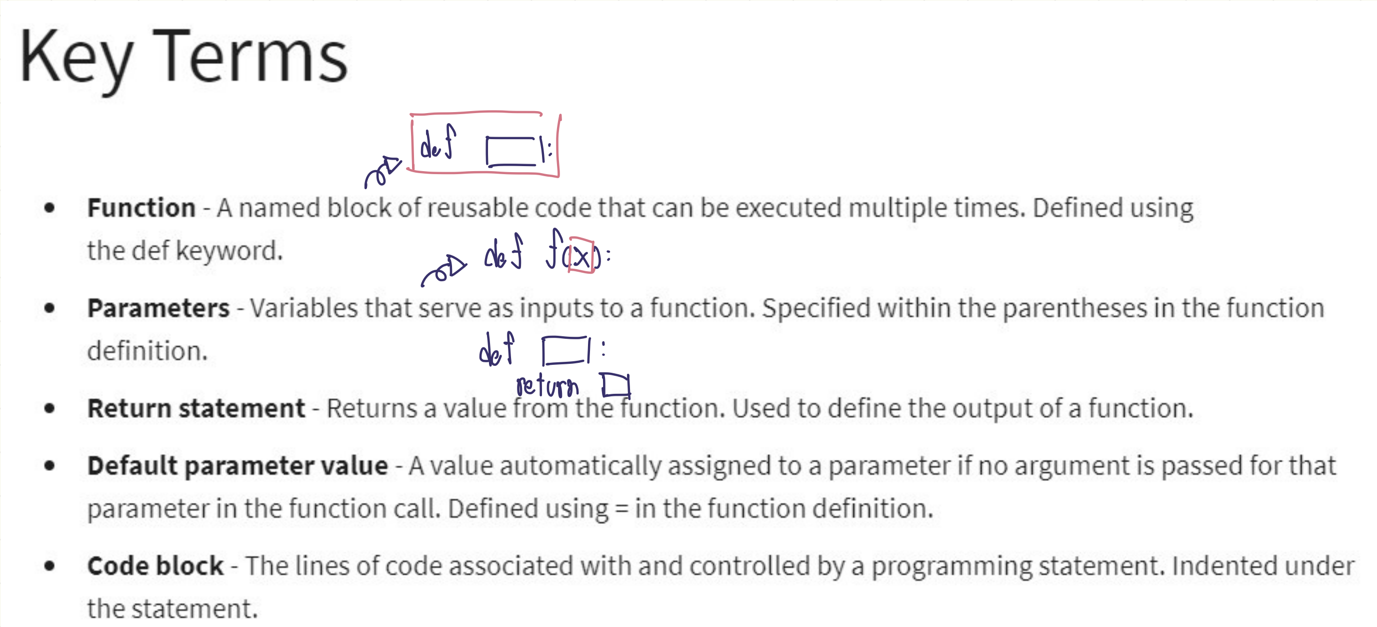


Key Terms

def □:

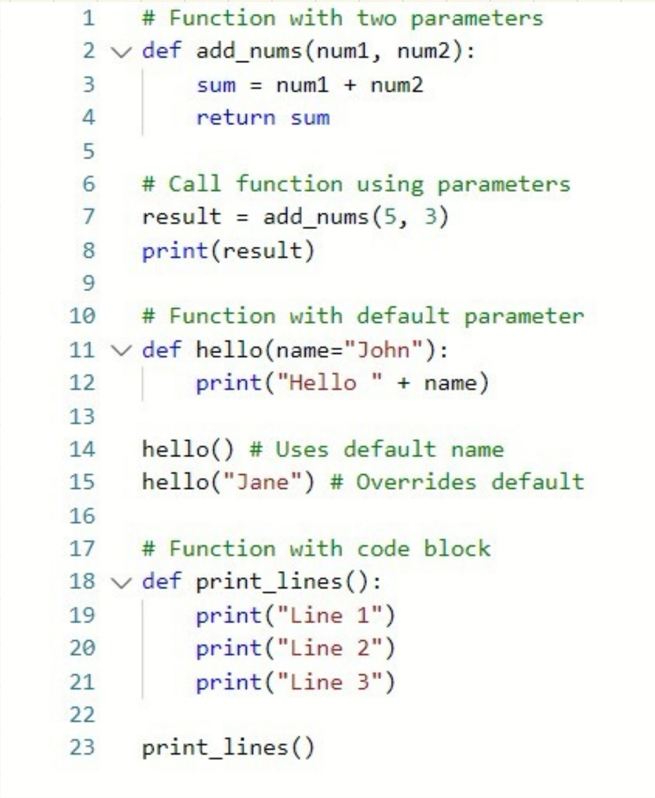
- **Function** - A named block of reusable code that can be executed multiple times. Defined using the def keyword.

def f(x):

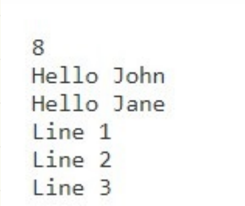
- **Parameters** - Variables that serve as inputs to a function. Specified within the parentheses in the function definition.

def □:
return □

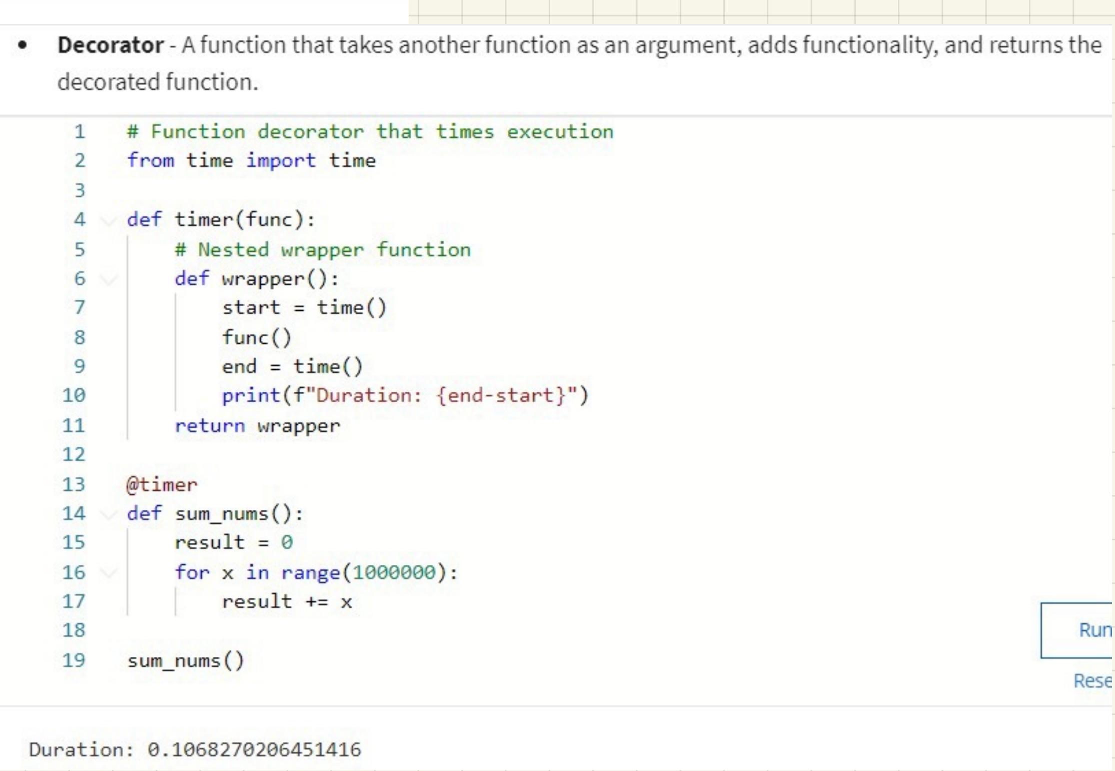
- **Return statement** - Returns a value from the function. Used to define the output of a function.
- **Default parameter value** - A value automatically assigned to a parameter if no argument is passed for that parameter in the function call. Defined using = in the function definition.
- **Code block** - The lines of code associated with and controlled by a programming statement. Indented under the statement.



```python
# Function with two parameters
def add_nums(num1, num2):
    sum = num1 + num2
    return sum

# Call function using parameters
result = add_nums(5, 3)
print(result)

# Function with default parameter
def hello(name="John"):
    print("Hello " + name)

hello() # Uses default name
hello("Jane") # Overrides default

# Function with code block
def print_lines():
    print("Line 1")
    print("Line 2")
    print("Line 3")

print_lines()
```



```
8
Hello John
Hello Jane
Line 1
Line 2
Line 3
```

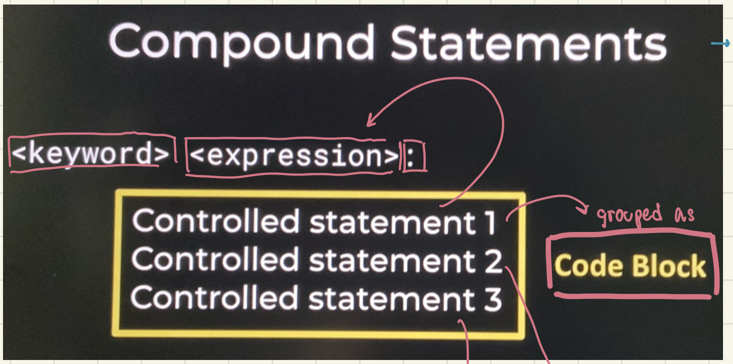


- **Decorator** - A function that takes another function as an argument, adds functionality, and returns the decorated function.

```python
# Function decorator that times execution
from time import time

def timer(func):
    # Nested wrapper function
    def wrapper():
        start = time()
        func()
        end = time()
        print(f"Duration: {end-start}")
    return wrapper

@timer
def sum_nums():
    result = 0
    for x in range(1000000):
        result += x

sum_nums()
```

```
Duration: 0.1068270206451416
```

Compound Statements in Python



Compound Statements

```
<keyword> <expression>:
    Controlled statement 1
    Controlled statement 2
    Controlled statement 3
```

grouped as **Code Block**

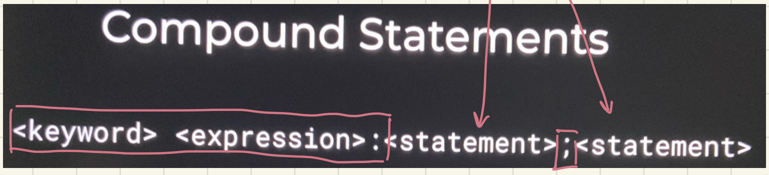
→ composed of a controlling statement and a group of control statements

↳ controlling statement determines the execution of controlled statements

are grouped in one of two ways

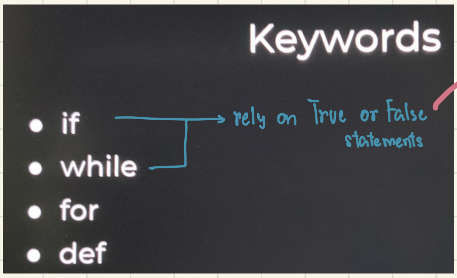
↳ common way : • group them as a Code Block
- sharing an indentation level

↳ group them in the same line separted by semicolons : • only done when they are brief
- easily read in a single line



Compound Statements

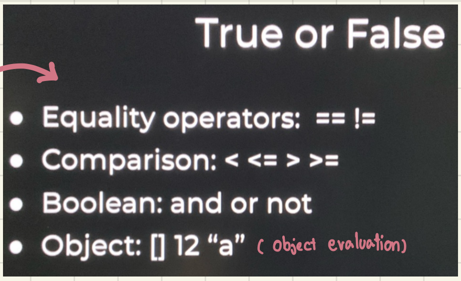
```
<keyword> <expression>:<statement>;<statement>
```



Keywords

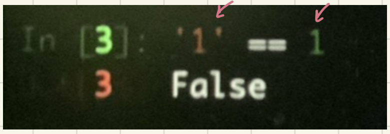
- if
- while
- for
- def

rely on True or False statements



True or False

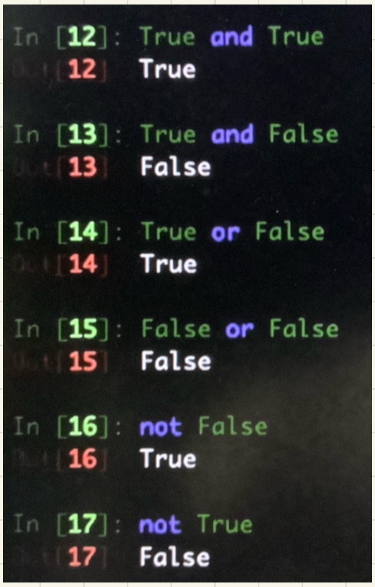
- Equality operators: == !=
- Comparison: < <= > >=
- Boolean: and or not
- Object: [] 12 “a” (object evaluation)



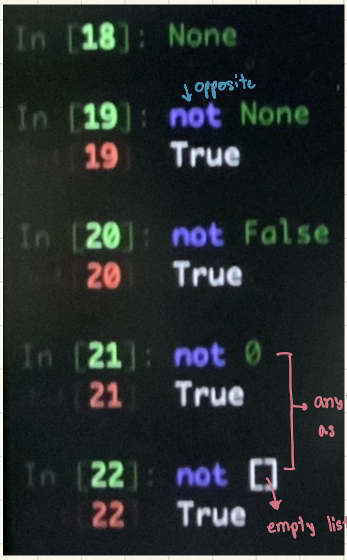
string numerical

```
In [3]: '1' == 1
Out[3]: False
```

Checking of Data Type



```
In [12]: True and True
Out[12]: True

In [13]: True and False
Out[13]: False

In [14]: True or False
Out[14]: True

In [15]: False or False
Out[15]: False

In [16]: not False
Out[16]: True

In [17]: not True
Out[17]: False
```



```
In [18]: None

In [19]: not None
Out[19]: True

In [20]: not False
Out[20]: True

In [21]: not 0
Out[21]: True

In [22]: not []
Out[22]: True
```

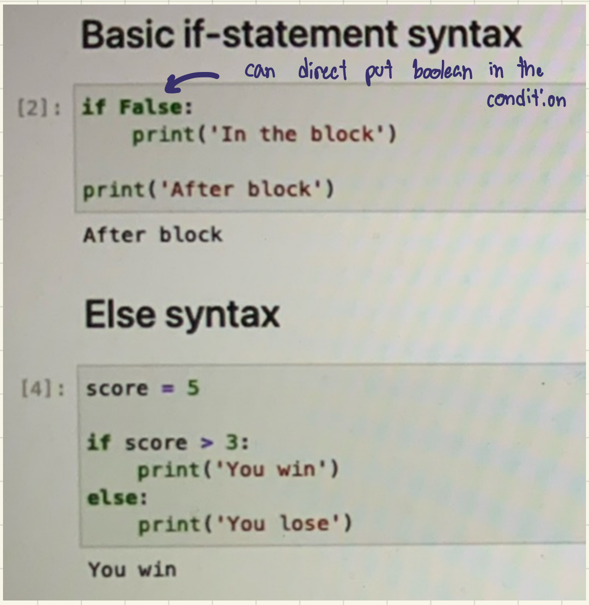
opposite

anything as zero

empty list = length of zero

anything has

If Statements in Python



Basic if-statement syntax

```
[2]: if False:
         print('In the block')

     print('After block')
```

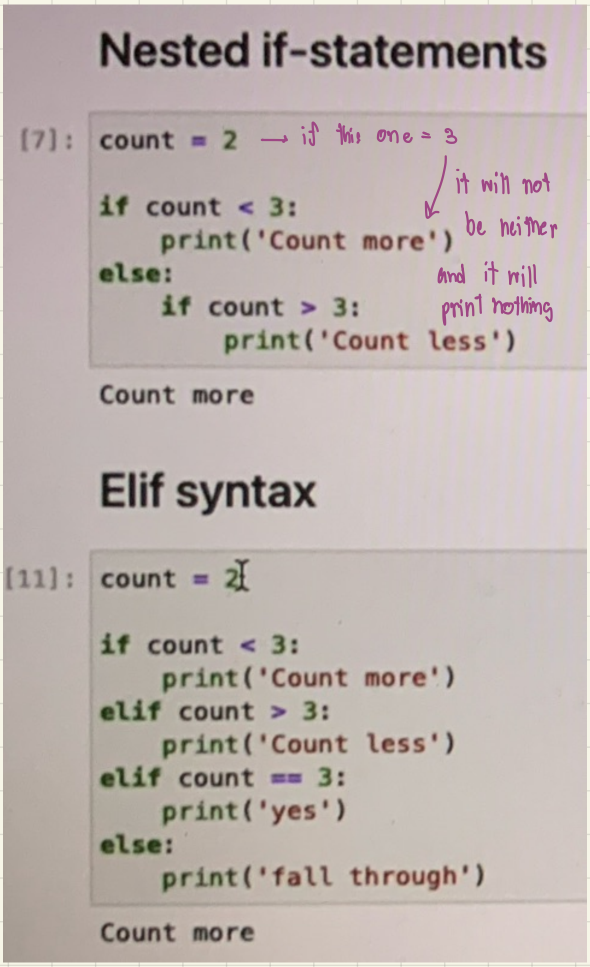
```
After block
```

can direct put boolean in the condition

Else syntax

```
[4]: score = 5

     if score > 3:
         print('You win')
     else:
         print('You lose')
```

```
You win
```



Nested if-statements

```
[7]: count = 2

     if count < 3:
         print('Count more')
     else:
         if count > 3:
             print('Count less')
```

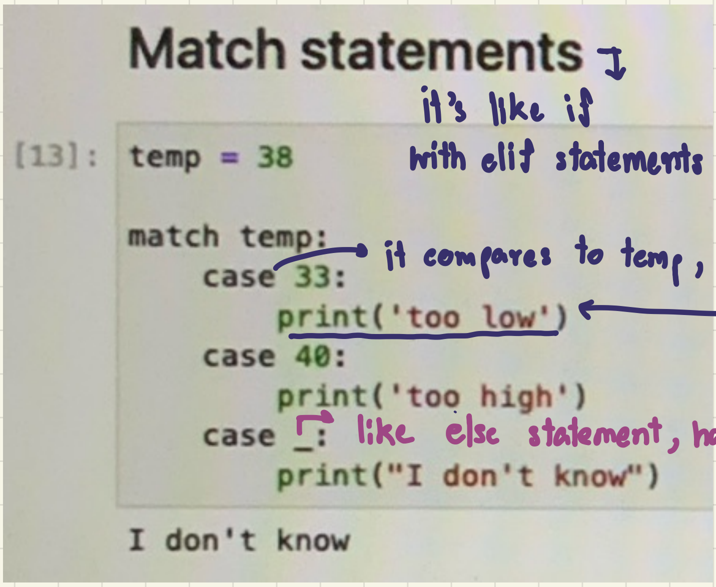
```
Count more
```

→ if this one = 3 it will not be neither and it will print nothing

Elif syntax

```
[11]: count = 2

      if count < 3:
          print('Count more')
      elif count > 3:
          print('Count less')
      elif count == 3:
          print('yes')
      else:
          print('fall through')
```

```
Count more
```



Match statements

it's like if with elif statements

→ New syntax in python 3.10+

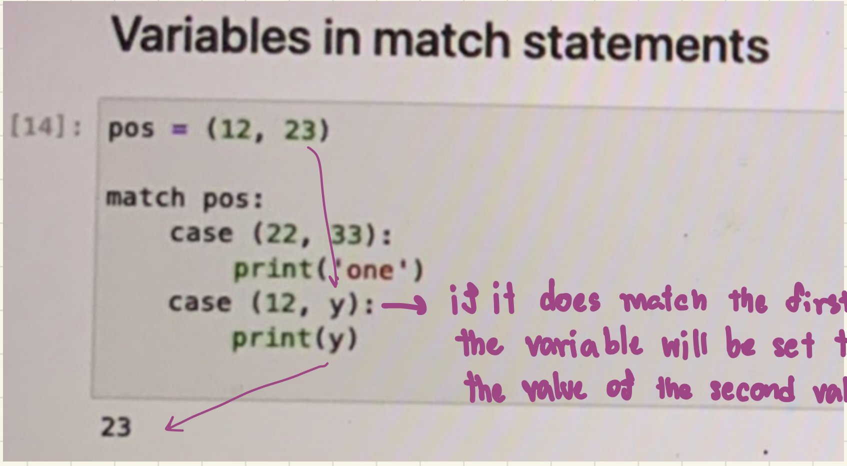
```
[13]: temp = 38

      match temp:
          case 33:
              print('too low')
          case 40:
              print('too high')
          case _:
              print("I don't know")
```

```
I don't know
```

it compares to temp, if temp equals to 33 then,

like else statement, happen when none of the other cases are True

One thing that separate match from if, elif



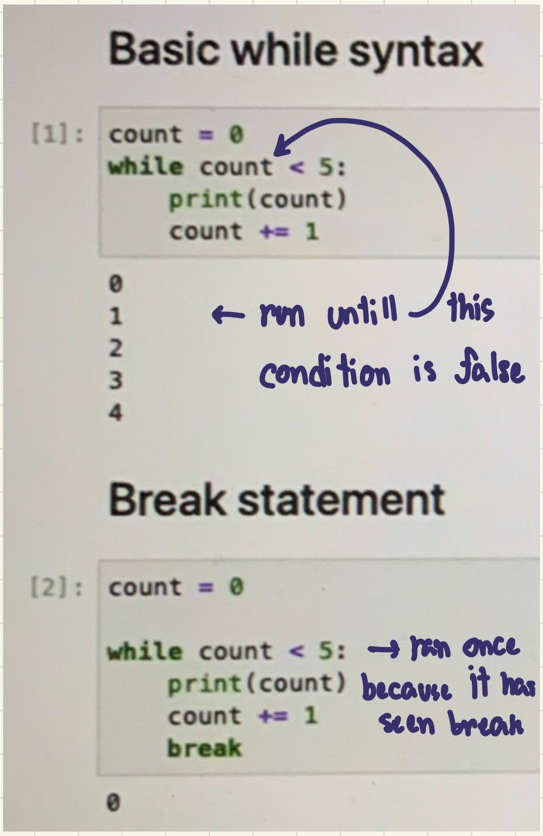
Variables in match statements

```
[14]: pos = (12, 23)

      match pos:
          case (22, 33):
              print('one')
          case (12, y):
              print(y)
```

```
23
```

if it does match the first value, the variable will be set to the value of the second value

While Loops in Python



Basic while syntax

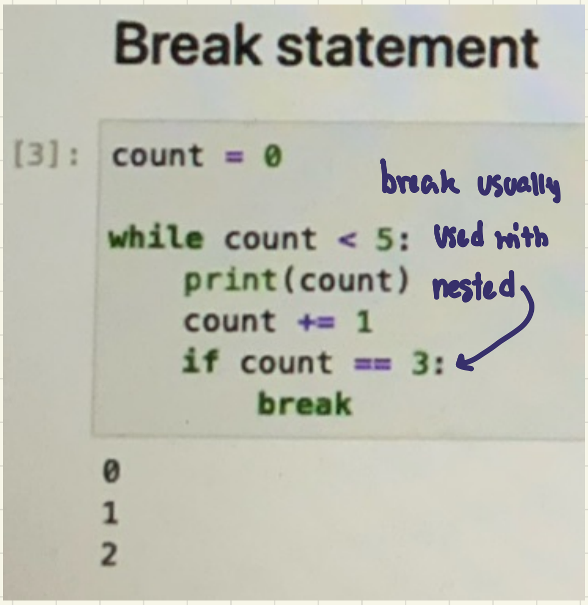
```
[1]: count = 0
     while count < 5:
         print(count)
         count += 1

0
1
2
3
4
```

← run untill this condition is false

Break statement

```
[2]: count = 0

     while count < 5:
         print(count)
         count += 1
         break

0
```

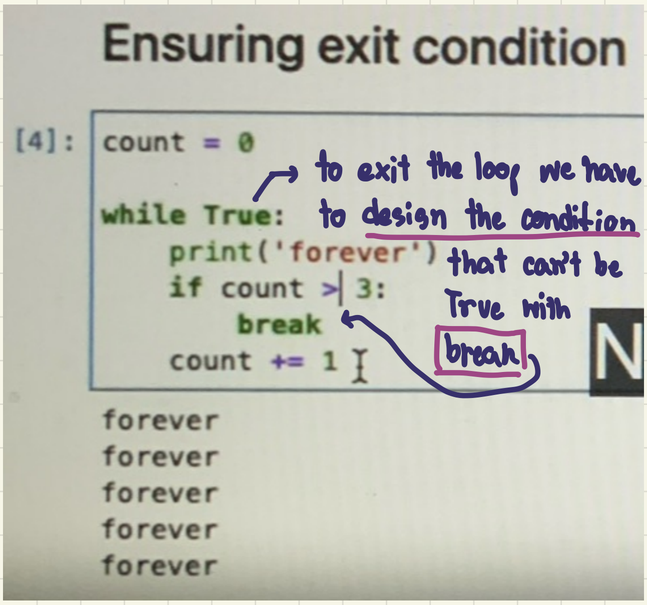
→ ran once because it has seen break



Break statement

```
[3]: count = 0

     while count < 5:
         print(count)
         count += 1
         if count == 3:
             break

0
1
2
```

break usually used with nested

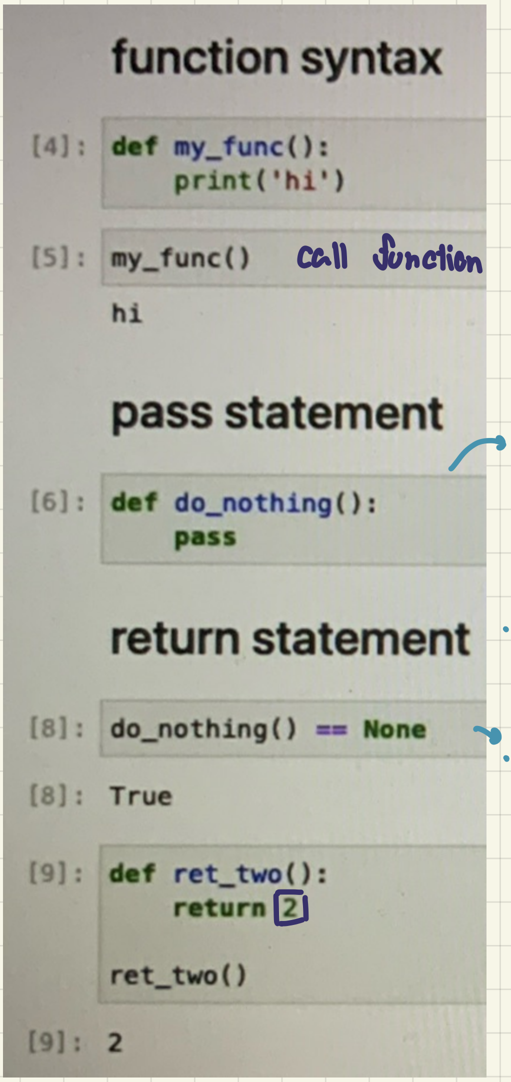


Ensuring exit condition

```
[4]: count = 0

     while True:
         print('forever')
         if count > 3:
             break
         count += 1

forever
forever
forever
forever
forever
```

to exit the loop we have to design the condition that can't be True with break

Functions in Python



function syntax

```
[4]: def my_func():
         print('hi')
```

```
[5]: my_func()
```

call function

```
hi
```

pass statement

```
[6]: def do_nothing():
         pass
```

return statement

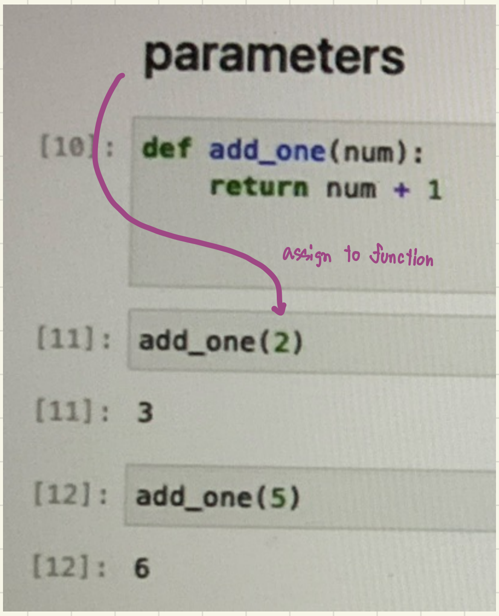
```
[8]: do_nothing() == None
[8]: True
```

```
[9]: def ret_two():
         return 2

     ret_two()
[9]: 2
```

- if you have already define your programe architecture, but this function not yet ready to implement their behavior
- it does nothins, hence it will be defined with out raising an error

(variable that not assign value yet)

- every thing in Python is equal to None

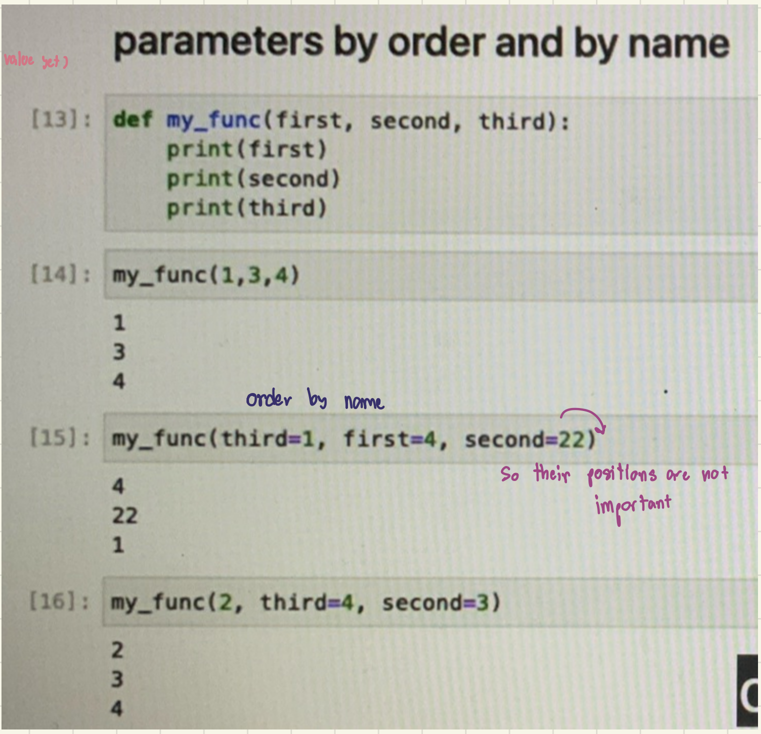


parameters

```
[10]: def add_one(num):
          return num + 1
```

assign to function

```
[11]: add_one(2)
[11]: 3
[12]: add_one(5)
[12]: 6
```



parameters by order and by name

```
[13]: def my_func(first, second, third):
          print(first)
          print(second)
          print(third)
```

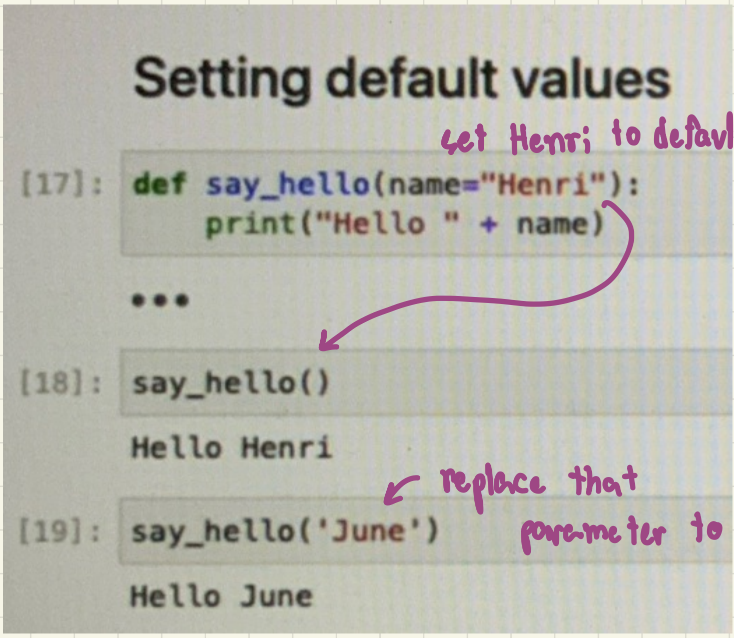
```
[14]: my_func(1,3,4)
      1
      3
      4
```

order by name

```
[15]: my_func(third=1, first=4, second=22)
      4
      22
      1
```

So their positions are not important

```
[16]: my_func(2, third=4, second=3)
      2
      3
      4
```



Setting default values

set Henri to default

```
[17]: def say_hello(name="Henri"):
          print("Hello " + name)
```

```
[18]: say_hello()
      Hello Henri
```

replace that parameter to June

```
[19]: say_hello('June')
      Hello June
```

python_decorator_functions

July 16, 2024

0.1 Timing Decorator

[1]:
```python
# Function decorator that times execution
from time import time

def timer(func):
    # Nested wrapper function
    def wrapper():
        start = time()
        func()
        end = time()
        print(f"Duration: {end-start}")
    return wrapper
```

import only time module in time package

[2]:
```python
@timer
def sum_nums():
    result = 0
    for x in range(1000000):
        result += x

sum_nums()
```

Call

func

← activate

print it was called

```
Duration: 0.05450153350830078
```

0.2 Logging Decorator

[33]:
```python
def logger(func):
    def wrapper(*args, **kwargs):
        print(f"Ran {func.__name__} with args: {args}, and kwargs: {kwargs}")
        return func(*args, **kwargs)
    return wrapper
```

[34]:
```python
@logger
def add(x, y):
    return x + y
```

Call

```
@logger
def sub(x, y):
    return x - y

add(10, 20)
sub(30, 20)
```

```
Ran add with args: (10, 20), and kwargs: {}
Ran sub with args: (30, 20), and kwargs: {}
```

[34]:

```
10
```

0.3 Caching Decorator

[35]:

```
import functools

def cache(func):
    cache_data = {}
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        key = args + tuple(kwargs.items())
        if key not in cache_data:
            cache_data[key] = func(*args, **kwargs)
        return cache_data[key]
    return wrapper
```

[36]:

```
import time
@cache
def expensive_func(x):
    start_time = time.time()
    time.sleep(2)
    print(f"{expensive_func.__name__} ran in {time.time() - start_time:.2f}␣
 ↪secs")
    return x
```

[37]:

```
%time print(expensive_func(1))
```

```
expensive_func ran in 2.00 secs
1
CPU times: user 10.4 ms, sys: 2.82 ms, total: 13.2 ms
Wall time: 2 s
```

[38]:

```
%time print(expensive_func(1))
```

```
1
CPU times: user 619 µs, sys: 100 µs, total: 719 µs
```

```
Wall time: 725 µs
```

[39]:

```
@cache
def fibonacci(n):
    if n < 2:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

[40]:

```
fibonacci(10)
```

[40]:

```
55
```

0.4 Delay

[41]:

```
import time
from functools import wraps

def delay(seconds):
    def inner(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(f"Sleeping for {seconds} seconds before running {func.
 ↪__name__}")
            time.sleep(seconds)
            return func(*args, **kwargs)
        return wrapper
    return inner
```

[42]:

```
@delay(seconds=3)
def print_text():
    print("Hello World")

print_text()
```

```
Sleeping for 3 seconds before running print_text
Hello World
```