Python Lecture 9 – Functions Advance (error handling, unknown arguments)

How to take unknown number of arguments in Functions?

You have used the print function and as you can see it can take literally any number of arguments. For example:

- print("Hello")
- print("Hello", name)

How can we tell the program that our function can take any number of arguments? Both, positional and named arguments.

We use the ***args** and ****kwargs** keywords.

What is *args?

Args is a shortcut name for positional arguments whose length is unspecified.

For example, you can create a sum function for any number of arguments.

```
def add(*numbs):

    ans = 0
    for num in numbs:
        ans += num
    return ans
```

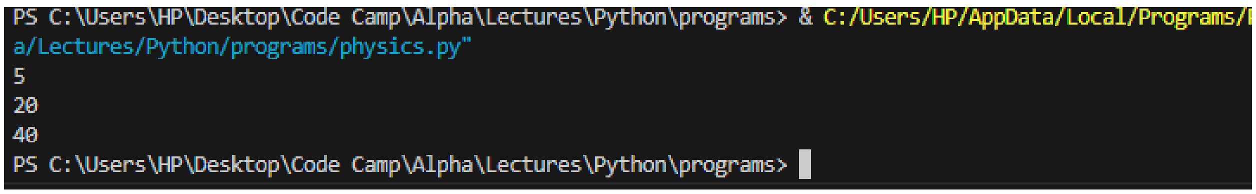
NOTE: Anything with single * is args. It is not mendatory to use the args word specifically. The syntax is to use the symbol * to take in a variable number of arguments; by convention, it is often used with the word args.

Code Camp | Alpha

Let's use it

```
print(add(2,3))
print(add(2,3,5,10))
print(add(2,3,5,10,20))
```

Check the outputs



```
PS C:\Users\HP\Desktop\Code Camp\Alpha\Lectures\Python\programs> & C:/Users/HP/AppData/Local/Programs/
a/Lectures/Python/programs/physics.py"
5
20
40
PS C:\Users\HP\Desktop\Code Camp\Alpha\Lectures\Python\programs>
```

NOTE:

- What *args allows you to do is take in more arguments than the number of formal arguments that you previously defined. With *args, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).
- Using the *, the variable that we associate with the * becomes iterable meaning you can do things like iterate over it.
- *args returns a tuple. Meaning that the values we put in as arguments becomes a tuple. So different iterable functions can be applied to them as well.

What is **kwargs?

**kwargs is similar to *args but instead of positional arguments, it can taky any number of named arguments.

NOTE:

- A keyword argument is where you provide a name to the variable as you pass it into the function.
- One can think of the kwargs as being a dictionary that maps each keyword to the value that we pass alongside it.
- Again, it is not mandatory to use kwargs but double **.

Example:

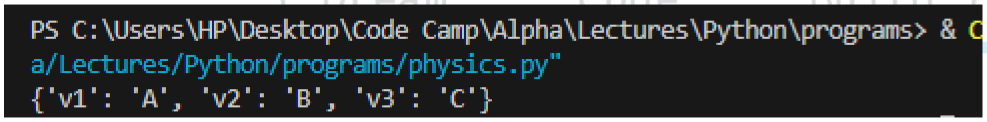
In the following example we have just created a function which can take any number of named arguments and print it. See in the output that a dictionary like structure is returned. (It is not the type dictionary though, but dictionary functions are applicable on it).

```
def my_func(**kwargs):
    print(kwargs)

my_func(v1="A", v2="B", v3="C")
```

Output:

{“Dream” “Code” “Build”}



```
PS C:\Users\HP\Desktop\Code Camp\Alpha\Lectures\Python\programs> & C
a/Lectures/Python/programs/physics.py"
{'v1': 'A', 'v2': 'B', 'v3': 'C'}
```

NOTE:

**kwargs also give us access to items, keys and values as well.

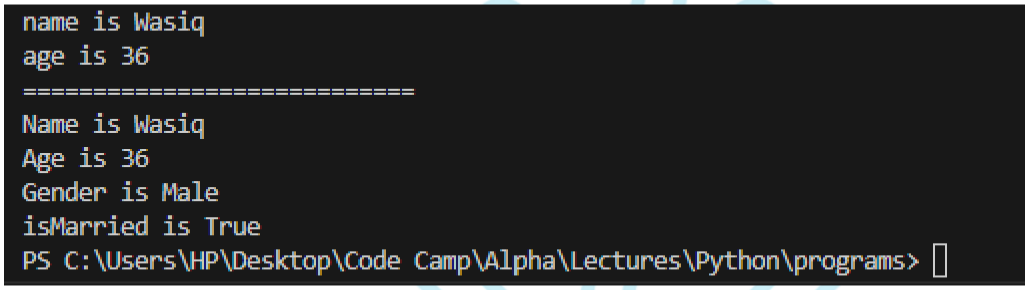
Example:

In the following example, we will try to access all keys and values

```python
def userdata(**data):
    for data_key, data_value in data.items():
        print(f"{data_key} is {data_value}")

userdata(name="Wasiq", age=36)
print("============================")
userdata(Name="Wasiq", Age=36, Gender="Male", isMarried=True)
```

Output:



```
name is Wasiq
age is 36
============================
Name is Wasiq
Age is 36
Gender is Male
isMarried is True
PS C:\Users\HP\Desktop\Code Camp\Alpha\Lectures\Python\programs>
```

Example:

In the following example we will access only values and do operations on it as well

```python
def summ(**numbers):
    res = 0
    for num in numbers.values():
        res += num
    return res

print(summ(a=5, b=10))
```

Code Camp | Alpha
{"Dream" , "Code" , "Build"}

Output:

15

Practical Use:

Now let's create a function called combined_resistance and based on the type of combination (either series or parallel) the function returns the equivalent resistance. The good part, user can resistances with any names, and can include any number of resistors. We will also use try / exception to implement it all.

Program without Error catching:

```python
def combined_resistance(combination, **resistances):
    """
    This is the docstring.
    This function will compute the equivalence resistance based on weather it is
series combination or parallel combination.
    User should define the type of connection first.
    User can input any number of resistances with any name. The program is
dynamic enough to work.
    """
    # Checking if no resistance is provided, the program should stop.
    if len(resistances) == 0:
        return print("Please provide values of resistances")

    # Now printing some headings and initial values.
    print("======================== \n Equivalence Resistance
\n========================")

    # Printing all the resistances
    print("\n".join([f"{key} = {value} ohms" for key,
          value in resistances.items()]))
    result = 0  # initializing result to zero.

    if combination.lower() == 'series':  # This part will run for series
combination
        for resistance in resistances.values():
            result += resistance
        return print(f"Equivalent {combination} Resistance is: {result} ohms")
    elif combination.lower() == 'parallel':  # This part will run for parallel
combination
        for resistance in resistances.values():
            result += 1/resistance
            output = 1/result
        return print(f"Equivalent {combination} is: {output} ohms.")
    else:
        return print("Wrong Selection")


combined_resistance(combination="Series", R1=30, R2=20, R3=40)
combined_resistance(combination="parallel", R1=5, R2=5)
```

Function with Error Catching:

```python
def combined_resistance(combination, **resistances):
    """
    This is the docstring.
    This function will compute the equivalence resistance based on whether it is
a series combination or parallel combination.
    Users should define the type of connection first.
    Users can input any number of resistances with any name. The program is
dynamic enough to work.
    """
    try:
        # Checking if no resistance is provided, the program should raise a
ValueError.
        if len(resistances) == 0:
            raise ValueError("Please provide values of resistances")

        # Now printing some headings and initial values.
        print(
            "========================= \n Equivalence Resistance
\n=========================")

        # Printing all the resistances
        print("\n".join([f"{key} = {value} ohms" for key,
              value in resistances.items()]))

        result = 0  # initializing result to zero.

        if combination.lower() == 'series':  # This part will run for series
combination
            for resistance in resistances.values():
                result += resistance
            print(f"Equivalent {combination} Resistance is: {result} ohms")
        elif combination.lower() == 'parallel':  # This part will run for
parallel combination
            for resistance in resistances.values():
                result += 1/resistance
            output = 1/result
            print(f"Equivalent {combination} is: {output} ohms.")
        else:
            raise ValueError("Wrong Selection of combination")

    except ValueError as e:
        print(f"Error: {e}")
```

```python
# Example usage
# combined_resistance('series', R1=10, R2=20, R3=30)
# combined_resistance('parallel', R1=10, R2=20, R3=30)
# combined_resistance('seRiEs', R1=30, R2=34)  # This will trigger the exception
```

We can uncomment and check the respective values.

A NOTE ON ERROR HANDLING:

In Python, errors and exceptions are broadly categorized into two main types: **syntax errors** and **exceptions**.

Syntax Errors:

Syntax errors, also known as parsing errors, occur when the code is not written according to the proper syntax of Python.

These errors are detected by the Python interpreter during the parsing phase before the code is executed.

Examples include missing colons, mismatched parentheses, or invalid indentation.

Exceptions:

Exceptions are runtime errors that occur when the program is being executed.

They are situations where the program encounters something unexpected during execution that it cannot handle automatically.

Examples of common exceptions include:

ValueError: Raised when a function receives an argument of the correct type but an inappropriate value.

TypeError: Raised when an operation or function is applied to an object of an inappropriate type.

ZeroDivisionError: Raised when division or modulo operation is performed with zero as the divisor.

FileNotFoundError: Raised when trying to open or access a file that doesn't exist.

IndexError: Raised when trying to access an index that is outside the range of a sequence (e.g., list, tuple).

It's important to note that exceptions are intended to be caught and handled by the programmer. This allows the program to gracefully recover from errors and continue execution, or to take appropriate action based on the type of error encountered.

Code Camp | Alpha
{"Dream", "Code", "Build"}

We can catch the errors using try / except / raise.

In Python, the try/except block is used for exception handling. Exceptions are also known as logical errors. They occur during a program's execution. Rather than allowing the program to crash when an error is detected, Python generates an exception you can handle. It's comparable to the way an iPhone displays a temperature warning when your phone gets too hot. Instead of allowing the phone to overheat, it stops itself from functioning and prompts you to resolve the issue by cooling it down. Similarly, Python exceptions halt the program and provide you with an opportunity to resolve the error instead of crashing. They are of two types:

1. User defined exceptions
2. Built – in exceptions

<u>try Block:</u>

The code that might raise an exception is placed inside the try block.

If an exception occurs in the try block, the control is transferred to the nearest except block.

<u>except Block:</u>

The except block catches and handles the exception raised in the try block.

It allows the program to gracefully handle errors without crashing.

<u>raise Statement:</u>

Raise() is a function that interrupts the normal execution process of a program. It signals the presence of special circumstances such as exceptions or errors. The raise statement is used to deliberately raise an exception during the execution of a program.

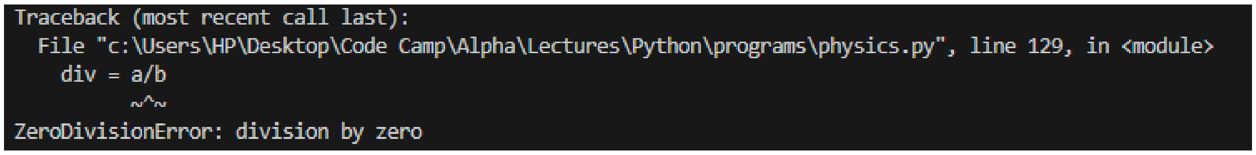
You can raise built-in exceptions or create custom exceptions.

<u>Example 1 – How to catch an exception generally?</u>

```
a = 5
b = 0

div = a/b
print(div)
```

The above code will give an error. Let us check:



```
Traceback (most recent call last):
  File "c:\Users\HP\Desktop\Code Camp\Alpha\Lectures\Python\programs\physics.py", line 129, in <module>
    div = a/b
          ~^~
ZeroDivisionError: division by zero
```

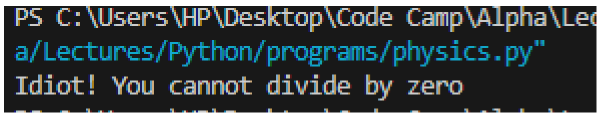
Now we can except this error as follows to display a custom message.

```
a = 5
b = 0

try:
```

```python
    div = a/b
    print(div)
except:
    print("Idiot! You cannot divide by zero")
```

Now if we run it. We can a custom exception here:



```
PS C:\Users\HP\Desktop\Code Camp\Alpha\Lec
a/Lectures/Python/programs/physics.py"
Idiot! You cannot divide by zero
```

This was a demonstration how to generally check for an exception. Now we will explore how to check specific exceptions.

Example 2 – How to check a specific exception?

We can target a particular type of error (mentioned above) and display custom messages. But for this happen we must know which types of errors might occur in our case. For example, in the above code, we may expect the user:

- May be dividing by zero (ZeroDivisionError)
- May have not introduced variables 'a' and 'b' already (NameError)
- May have assigned non – numeric forms to a and b (TypeError)

In such cases, you can display a different message in each type of error in the exceptions.

```python
a = 5
b = 'zero'

try:
    div = a/b
    print(div)
except ZeroDivisionError:
    print("Idiot! You cannot divide by zero")
except TypeError:
    print("You must convert strings to floats or integers before dividing")
except NameError:
    print("A variable you're trying to use does not exist")
```

Can you guess the error that will be displayed?

{"Dream" , "Code" , "Build"}

Similarly, we can handle multiple types of exceptions at once. For example:

```python
a = 5
b = 'zero'

try:
    div = a/b
    print(div)
except (ZeroDivisionError, TypeError):
    print("This division is impossible")
```

NOTE:

That is not mandatory to only use print statements in except blocks, you can run any code in it.

```
a = 5
b = 0

try:
    div = a/b
    print(div)
except ZeroDivisionError:
    b = 0.1
    div = a/b
    print(div)
    print("Issue is fixed!")
```

Here in the case of reaching a ZeroDivisionError, we have changed the value of b to be a non – zero and resolve the issue.

Suppose you want to implement a few special cases but handle all other exceptions the same. In that case, you can create an except block to handle all other exceptions:

```
a = 5
b = 0

try:
    div = a/b
    print(div)
except (ZeroDivisionError, TypeError):
    print("You cannot divide by zero and variables must be floats or integers")
except:
    print("Other error")
```

Code Camp | Alpha

Example 3 – How to raise an exception?

The difference between except and raise is just like the difference between receiving a call (exception) and making a call (raise). Exceptions only run when they occur in a program. However, if we want to throw an exception, we have to use raise. For example, if you asked user to pick time between 9am to 5pm and the user inputs a time not between this range. Here we can throw an error. (Some programming languages use the throw keyword, python use raise). It's like using conditions and displaying an error.

{"Dream", "Code", "Build"}

For example:

```
time = int(input("Enter time between 9 to 5: "))
if time < 5 or time > 9:
    raise ValueError("This is not the time you have been asked for")

print("This is the normal flow of the program")
```

In the code block above, ValueError is the specified error. It has been set to occur anytime the variable time is not in the given range. The text inside the parentheses represents your chosen text to print to the user.

You may wonder why show error when we simply can print the message? It is because printing will not stop the program, errors will. We will raise an error to show if something of very critical nature may create issues in some later part of the program.

Note: We can also raise a general exception.

```
x = -5

if x < 0:
    raise Exception("Sorry, please enter a number greater than or equal to 0")
```

This will through an exception when the value of x is negative.

Printing an Error!

If you are not convinced to display an error with its complexity, you can also print it like normal printing.

Let’s modify our division example,

```
a = 5
b = 0

try:
    div = a / b
    print(div)
except Exception as X:
    print(X)
```

In this example, the error message will be displayed as it is normally printed. Here normal Exception Error will be displayed.

Code Camp | Alpha

Also we can except any specific error and display it as message as well.

```
a = 5
b = 0

try:
    div = a / b
    print(div)
except ZeroDivisionError as X:
    print(X)
```

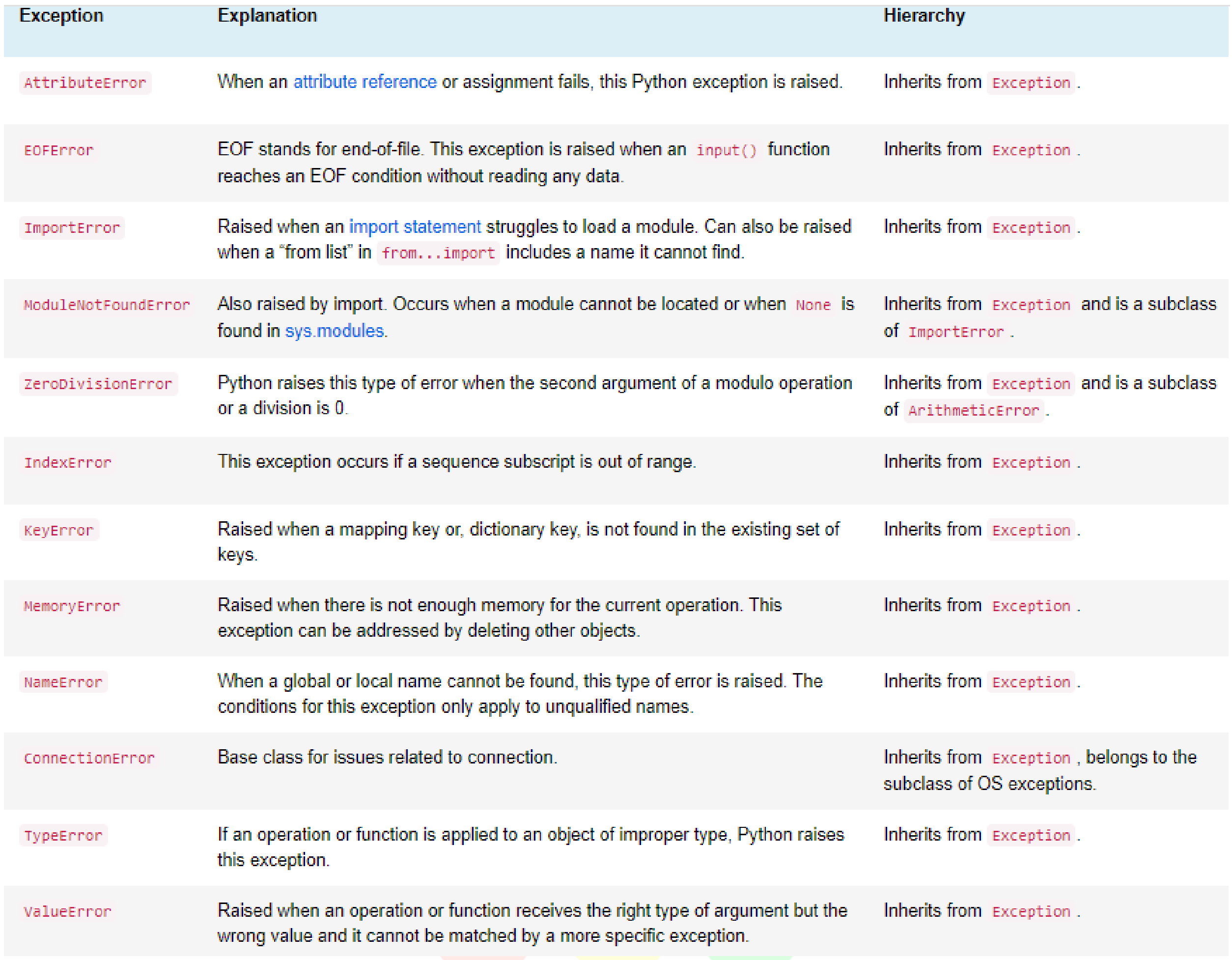
Or may be with an error message.

```
except ZeroDivisionError as X:
    print(f"Error: {X}")
```

Official Documentation:

https://docs.python.org/3/tutorial/errors.html

TYPES OF EXCEPTIONS SUMMARY



Exception	Explanation	Hierarchy
`AttributeError`	When an attribute reference or assignment fails, this Python exception is raised.	Inherits from `Exception`.
`EOFError`	EOF stands for end-of-file. This exception is raised when an `input()` function reaches an EOF condition without reading any data.	Inherits from `Exception`.
`ImportError`	Raised when an import statement struggles to load a module. Can also be raised when a "from list" in `from...import` includes a name it cannot find.	Inherits from `Exception`.
`ModuleNotFoundError`	Also raised by import. Occurs when a module cannot be located or when `None` is found in sys.modules.	Inherits from `Exception` and is a subclass of `ImportError`.
`ZeroDivisionError`	Python raises this type of error when the second argument of a modulo operation or a division is 0.	Inherits from `Exception` and is a subclass of `ArithmeticError`.
`IndexError`	This exception occurs if a sequence subscript is out of range.	Inherits from `Exception`.
`KeyError`	Raised when a mapping key or, dictionary key, is not found in the existing set of keys.	Inherits from `Exception`.
`MemoryError`	Raised when there is not enough memory for the current operation. This exception can be addressed by deleting other objects.	Inherits from `Exception`.
`NameError`	When a global or local name cannot be found, this type of error is raised. The conditions for this exception only apply to unqualified names.	Inherits from `Exception`.
`ConnectionError`	Base class for issues related to connection.	Inherits from `Exception`, belongs to the subclass of OS exceptions.
`TypeError`	If an operation or function is applied to an object of improper type, Python raises this exception.	Inherits from `Exception`.
`ValueError`	Raised when an operation or function receives the right type of argument but the wrong value and it cannot be matched by a more specific exception.	Inherits from `Exception`.

Errors Cheat sheet: https://www.coursera.org/tutorials/python-exception-cheat-sheet

Code Camp | Alpha
{"Dream" , "Code" , "Build"}