Python Fundamentals

Call stack, exceptions, try and except clauses, raise keyword

Objectives:

- Call Stack
- Handling Exceptions
  - try and except keywords
- Raising exceptions
  - raise keyword

Call Stack

- When running an application, a list of method calls is constantly maintained for the current statement that is being executed.

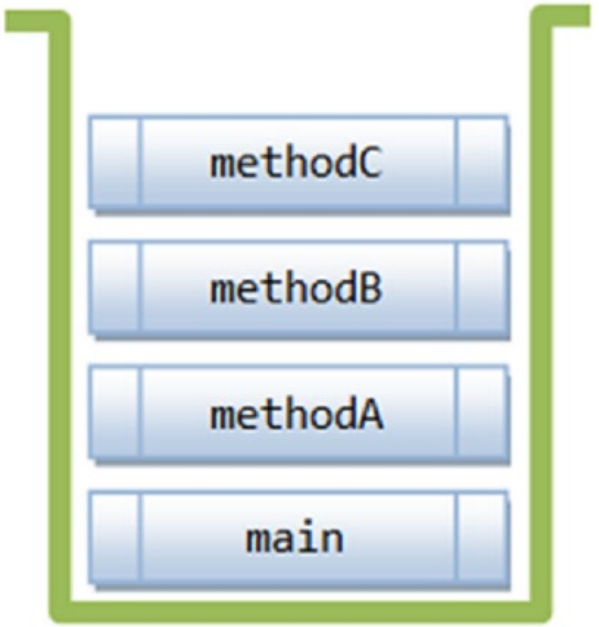
- This list of method calls starts with the very first method that was called, then the method it called, then the method **that** method called, and so on, until we reach the current statement.

- This is called the **call stack**

Call Stack

The picture to the right is showing an example of a call stack. Specifically, that:

1. **main** was called
2. **main** called **methodA**
3. **methodA** called **methodB**
4. **methodB** called **methodC**



Exceptions

- An exception is an object that is created and "raised" when an error or something unexpected occurs.
- An exception can be detected and "handled". This enables a program to choose how to deal with the situation.
- In order to apply Exception handling we need to implement **try** and **except** clauses.

Exceptions

- If an exception is not handled by the method in which it occurred, Python will look at the method that called that method.
- If **that** method doesn't handle the exception, it will keep working its way through the call stack.
- If the exception reaches the starting point of your application and isn't handled, your application will crash!

try keyword

- To handle an exception, we place a **try clause** around statements that could potentially cause an exception to be thrown.

- If at any point a statement within the **try clause** causes an exception to occur, it will **immediately** leave the clause **without** executing any remaining statements within the clause.

- If no exception occurs, our code will proceed as normal.

except keyword

- A try clause should be accompanied by one or more **except clauses**.
- Each except clause is designed to handle exceptions of a specific type. This is specified as a type for the **except clause.**
- Python will look for an except clause with the matching exception type. If found, execution *immediately* gets passed to that except clause.

Handling exceptions example

```
string_age = "Twenty"

try:
    int_age = int(string_age)
    print(int_age)
except ValueError:
    print("Age must be a valid number!")

print("End")
```

Without the try - except clauses, the error will cause the program to crash

If no exception occurs, the program will print age and then the word “end”

The exception causes the try clause to terminate.
The except clause determines what happens next.

Validation

…
Enter name: **Duy**
Enter age: **21**
Enter address: **16 Main St**
…

Validation

```
...
Enter name:                 Duy
Enter age:                  twenty one
        Invalid age. Please re-enter.
Enter age:
```

Validation

…
Enter name: **Duy**
Enter age: **twenty one**
Invalid age. Please re-enter.
Enter age: **21.4**
Invalid age. Please re-enter.
Enter age:

Validation

…
Enter name: **Duy**
Enter age: **twenty one**
Invalid age. Please re-enter.
Enter age: **21.4**
Invalid age. Please re-enter.
Enter age: **21**
Enter address: **16 Main St**
…

Validation iteration for age

```
…
while not is_age_valid:
    try:
        age_string = input("Enter age: ")
        age = int(age_string)
        is_age_valid = true
    except ValueError:
        print("Invalid age. Please re-enter.")
…
```

Design for try/except clauses

When there is a possibility that an exception will be raised:

- All the code to be executed if *no exception* is raised goes in a **try** clause. This is your happy path.
- Code to be executed *if* an exception is raised goes in a **except** clause.
- Code which is to be executed regardless of whether an exception is raised goes after the last **except** clause.

Design for try/except clauses

When writing an except clause, you should **ONLY** do the following:

- Log that an error occurred
- Provide a mechanism to report the error to the user
- Clean up any intermediate results; operations that were partially completed because of the exception being raised.

raise keyword

We want to create a program to validate if a number is a valid integer number and also if it is greater than 25 and less than 1000.

In case the number is not valid, we want the program to raise a **ValueError**.

raise keyword

```
try:
    …
    x = int(input_string)
    if x >= 1000:
        ex = ValueError()
        raise ex
    if x <= 25:
        ex = ValueError()
        raise ex
except ValueError:
   error = "Must be an integer "
            + " and must be greater than 25 "
            + " and must be less than 1000 "
   print(error)
```

Alternative syntax

```
try:
    …
    x = int(input_string)
    if x >= 1000:
        raise ValueError()
    if x <= 25:
        raise ValueError()
except ValueError:
   error = "Must be an integer "
            + " and must be greater than 25 "
            + " and must be less than 1000 "
   print(error)
```

Exceptions: except and raises

When an exception is raised (keyword: **raise**) anywhere in a method, it can be:

- Caught in that method (try, except clauses)
- Passed to the method that called our method. Or in other words, raised to the next method in the *call stack*.

Don’t forget that an unhandled exception will eventually crash our application!

Example

```
while True:
    try:
        age_string = input(prompt)
        age = to_age(age_string)
        break
    except ValueError as ex:
        print(tryAgain)
```

to_age()
is a function
that converts
and validates
a string that
represents
an age.

If the string
cannot be
converted,
we want
to_age()
to raise an
exception.

Example

```
def to_age(s: str) -> int:
      a = int(s)
      return a
```

If `int()` raises an exception, the program will crash because the ValueError is not caught

Handle exception

```
def to_age(s: str) -> int
    try:
        a = int(s)
        return a
    except ValueError as ex:
        ...
```

This is unsatisfactory because the method that calls `to_age()` wants to handle this exception.

Exceptions

```
def to_age(s: str) -> int:
    a = int(s)
    return a
```

If `int()` raises a ValueError,
`to_age()` now immediately raises the exception
back to the method that called `to_age()`.

Raising exceptions

- The body of the `int()` function creates the ValueError object and raises it with the **raise** keyword.

- Instead of handling it, the `int()` function allows the exception to be passed to the calling method.

- In this example, the `int()` function will raise the ValueError back to the `to_age()` function. The `to_age()` method will then raise it back to where `to_age()` was called.

Example 1

```
def main():
    …
    func_a()
    …
```

```
def func_a():
    …
    func_b()
    …
```

```
def func_b():
    …
    raise MyEx
    …
    except MyEx
    …
```

Example 2

```
def main():
    …
    func_a()
    …
```

```
def func_a():
    …
    func_b()
    …
    except MyEx
    …
```

```
def func_b():
    …
    raise MyEx
    …
```

Example 3

```
def main():
    …
    func_a()
    …
    except MyEx
    …
```

```
def func_a():
    …
    func_b()
    …
```

```
def func_b():
    …
    raise MyEx
    …
```

Debugging hint

If you are ever unsure why an exception is being raised try displaying the message in the exception object.

Every exception also records and can display what's known as a **stack trace**. This records the method in which the original exception was raised, then all the method calls up to the point at which the exception was handled.

Understanding the **stack trace** is very helpful when trying to isolate where an exception occurred.

Stack Trace Example

```
Traceback (most recent call last):
  File "...\program.py", line 19, in <module>
    number = to_int(input_text)
  File "...\program.py", line 16, in to_int
    return int(s)
 ValueError: invalid literal for int() with base
10: 'Twenty'
```

In the above example our exception was raised in the **to_int** function.

References

Python tutorial on handling errors

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

Demonstration:

- Call Stack
- Handling Exceptions
  - try and except keywords
- Raising exceptions
  - raise keyword