SDEV 1001

Programming Fundamentals

More Loops and Exceptions - 3

Expectations - What I expect from you

- No Late Assignments

- No Cheating

- Be a good classmate

- Don’t waste your time

- Show up to class

WE ARE ESSENTIAL TO ALBERTA | NAIT

Agenda

On the right is what we will cover today.

- Introduction to Exceptions in Python
- What Happens Without Exception Handling?
- Using try-except to Handle Errors
- Handling Multiple Exceptions
- Using else and finally
- Throwing (Raising) Exceptions in Python
- Best Practices
- Summary

Introduction to Exceptions in Python

- Exceptions are errors that occur during program execution.
- Common exceptions: `NameError`, `TypeError`, `ValueError`, `ZeroDivisionError`, `IndexError` etc.
- Handling exceptions prevents your program from crashing and allows you to respond gracefully.
- Exceptions can be caught and handled using `try` and `except` blocks.
- You can also throw exceptions using the `raise` keyword.

What Happens Without Exception Handling?

In Python, if an error occurs and you don't handle it, your program will crash.

- If the user enters `"twenty"` to the input, the program crashes with a `ValueError`. That's because you tried to convert a non-numeric string to an integer with `int`.

```
age = input("Enter your age: ")
years = 100 - int(age)
print(f"You will be 100 in {years} years.")
```

Here's what the output looks like in the terminal if this is your file (named `test_example.py`):

```
$ python test_example.py
Enter your age: twenty
Traceback (most recent call last):
  File "C:\Users\dmouris\temp\test_example.py", line 2, in <module>
    years = 100 - int(age)
                  ^^^^^^^^
ValueError: invalid literal for int() with base 10: 'twenty'
```

- The program stops running, and you see an error message in the terminal, sometimes you don't want this to happen and this is where you want to use exception handling.

Using try-except to Handle Errors

Let's update the previous example to handle the error gracefully using a `try` and `except` block.

- Here in the `try` block, we attempt to convert the input to an integer. If it fails, we catch the `ValueError` in the `except` block and print a friendly message instead of crashing.

```
age = input("Enter your age: ")
try:
    years = 100 - int(age)
    print(f"You will be 100 in {years} years.")
except ValueError as e:
    print(f"Invalid input! Please enter a number. ({e})")
```

Here's what the output looks like in the terminal if this is your file (named `test_example.py`):

```
$ python test_example.py
Enter your age: twenty
Invalid input! Please enter a number. (invalid literal for int() with base 10: 'twenty')
```

Handling Multiple Exceptions

Example of handling multiple exceptions in a single `try` block

You can handle multiple different exceptions in a single `try` block by using multiple `except` clauses.

```python
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)
except ValueError:
    print("That's not a valid number!")
except ZeroDivisionError:
    print("You can't divide by zero!")
```

This code handles both `ValueError` (if the input is not a number) and `ZeroDivisionError` (if the user tries to divide by zero).

Handling Multiple Exceptions

Some sample outputs based on the example on the last slide:

Let’s take a look at the output if you run this code and enter `0` as the input:

```
$ python test_example.py
Enter a number: 0
You can't divide by zero!
```

Let’s take a look at the output if you run this code and enter `twenty` as the input:

```
$ python test_example.py
Enter a number: twenty
That's not a valid number!
```

Using else and finally

There's also an `else` block that runs if no exceptions occur, and a `finally` block that always runs, regardless of whether an exception was raised or not.

- `else` runs if no exception occurs.
- `finally` always runs, whether there was an error or not.
- Note I recommend avoiding using `else` and `finally` unless you have a specific reason to use them, as they can make your code harder to read.

```python
try:
    # Code that might raise an exception
    print("Trying something risky...")
except Exception:
    print("An error occurred.")
else:
    print("No errors occurred!")
finally:
    print("This always runs, error or not.")
```

Throwing (Raising) Exceptions in Python

- Sometimes you want to signal that an error has occurred in your own code.
- You can do this by "raising" an exception using the `raise` keyword.
- This is a bit more advanced, but it’s useful for creating custom error handling, especially in larger applications.

Example:

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero!")
    return a / b

try:
    result = divide(10, 0)
except ValueError as e:
    print(f"Error: {e}")
```

Throwing (Raising) Exceptions in Python

Here's the output of the last slide

Here's what the output looks like in the terminal if this is your file (named `test_example.py`):

```
$ python test_example.py
Error: Cannot divide by zero!
```

Best Practices

- Keep the code inside `try` blocks as small as possible.
- Handle only the exceptions you expect.
- Inform the user about what went wrong.
- Use specific exception types instead of a general `Exception` when possible.

Summary

- Exceptions help you manage errors and keep your programs running smoothly.
- Use `try`, `except`, `else`, and `finally` to handle errors and clean up.
- Use `raise` to throw exceptions when needed.
- Practice handling exceptions to make your code more robust!

NAIT

Example

Let’s go run a few examples together