Sentinel

Exceptions and Error Handling

DEFENDING OUR DIGITAL WAY OF LIFE

Exceptions

- Exceptions

  unexpected events that disrupt the normal flow of a program's execution.

```
>>> number = input('enter a number')
>>> print(1 / int(number))
```

- What happens if the user enters 0?

```
ZeroDivisionError: division by zero
```

Common exceptions

- IndexError

  Trying to access an index out of range

```
>>> l = [1, 2, 3]
>>> l[3]
IndexError: list index out of range
```

- TypeError

  Operation is not compatible with the type

```
>>> x = 3 + 'hello'
TypeError: unsupported operand type(s)
for +: 'int' and 'str'
```

- NameError

  Python doesn't know the variable name

```
>>> print(asdf)
NameError: name 'asdf' is not defined
```

- ZeroDivisionError

Handling exceptions

- Exceptions are unexpected - we can't know for sure they will happen.
- As programmers, we have to prepare for them.
- We should *handle* them:

  Tell the program what to do when an exception occurs
- Let's see how to do it.

try-except

- like if-else for errors.
- *try* - where the standard logic is.
- *except* - what happens when an exception occurs.

```
# get a number from the user
number = input('enter a number')

try:
    print(1 / int(number))
except ZeroDivisionError:   # handle the exception
    print('0 is not a valid option')
```

Handling multiple exceptions

- You can handle multiple exceptions in the same try-except block.

```
# get a number from the user
number = input('enter a number')

try:
    print(1 / int(number))
except ZeroDivisionError:
    print('0 is not a valid option')
# what if the user enters a non-numeric string?
except ValueError:
    print('input must be a number')
```

else

- Tells the program what to do if there was no exception at all:

```
# get a number from the user
number = input('enter a number')

try:
    print(1 / int(number))
except ZeroDivisionError:
    print('0 is not a valid option')
except ValueError:
    print('input must be a number')
# executes if there was no exception
else:
    print('everything is fine')
```

finally

- Tells the program what to do anyway - if there was an exception or not.
- Used for cleanups (i.e making sure that files are closed)

```
# get a number from the user
number = input('enter a number')

try:
    print(1 / int(number))
except ZeroDivisionError:
    print('0 is not a valid option')
except ValueError:
    print('input must be a number')
# executes if there was no exception
else:
    print('everything is fine')
finally:
    print('this will be printed, no matter what')
```

Generic exceptions

- You can catch more than one type of exception using the generic Exception:

```
# get a number from the user
number = input('enter a number')

try:
    print(1 / int(number))
except Exception:  # will catch any error
    print('an error has occurred')
```

- You should try to catch exceptions as specific as possible - try to avoid using the generic one.

Catch and print the exception

- You can get the exception object in a variable, and use it to print it as a string:

```
# get a number from the user
number = input('enter a number')

try:
    print(1 / int(number))
except Exception as e:
    print(e)  # prints the exception as a string
```

Why handle exceptions?

- Prevent the program from crashing
- User experience
  - Smooth run instead of cryptic error messages
- Security
  - Unhandled exceptions can be exploited by hackers
  - Potential security problem

Tracebacks

- Printed when an exception is not handled.
- Shows in which function and module the exception occurred.
- How to read a traceback:

```
Traceback (most recent call last):
  File "example.py", line 10, in <module>
    result = divide_numbers(10, 0)
  File "example.py", line 3, in divide_numbers
    return x / y
ZeroDivisionError: division by zero
```

the line that called this function

the line number, and in which function and file

The line that cause it

The exception

Raising exceptions

- You can raise exceptions yourself, intentionally
- Why raise exceptions?

  To signal that something unexpected has occurred, requiring special handling.
- How to raise exceptions?

  using ‘raise’

raise

- Used to raise exceptions.
- Example:

```
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
    else:
        print(f"Age set to {age}")
```

```
try:
    set_age(-5)
except ValueError as e:
    print(e)  # Output: Age cannot be negative
```

Custom exceptions

- You can create your own custom exceptions
- Create a new class with the name of the exception
- Inherit from Exception
- Example:
  - __str__ should return the string that will be printed when the error is raised

```
class NegativeAgeError(Exception):
    def __str__(self):
        return "Age cannot be negative"
```

```
def set_age(age):
    if age < 0:
        raise NegativeAgeError()
    else:
        print(f"Age set to {age}")
```

Best practices

- Catch specific exceptions

  So you don't catch by mistake exceptions that you're not prepared to handle.

- Minimize code inside the "try" block

  So it's easier to identify the source of the error

- Keep these practices so you don't create hard-to-find bugs!

What did we learn?

- What exceptions are
- Common exceptions
- How to handle exceptions
  - try-except-else-finally
- How to read tracebacks
- How to raise exceptions
- How to create custom exceptions
- Best practices