Errors And Exceptions

The Basic Idea

An **exception** is a way to interrupt the normal flow of code.

Think about how ordinary code progresses from one line to the next... that's the normal flow.

An exception lets you jump out of that normal flow. So you can handle an error, or another exceptional situation.

Try/Except

Function calls, and other Python code, can **raise** an exception.

Example: `int()` takes a string. But a bad string raises `ValueError`.

```
>>> int("42")
42
>>> int("not a number")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'not a number'
```

Code can catch `ValueError` and take alternative action:

```
user_input = input("Pick a number from 1 to 10: ")

try:
    value = int(user_input)
except ValueError:
    value = 0

print(f"Got value: {value}")
```

Exceptional JSON

```
import logging
import requests
from json.decoder import JSONDecodeError

def fetch_object(api_url):
    resp = requests.get(api_url)
    try:
        # Parse response body as JSON
        obj = resp.json()
        logging.debug("Received object from url %s : %r", api_url, obj)
    except JSONDecodeError:
        # Response does not encode a valid JSON object.
        obj = {}
        logging.error("Response is not valid JSON: %s", api_url)
    return obj
```

Exceptions Are Pinpointed

An **exception** can be **raised** at any point. Even in the middle of a line.

```
def digits():
    print("CALLING: digits()")
    return "42"
def words():
    print("CALLING: words()")
    return "forty two"
```

```
>>> int(digits()) + int(words())
CALLING: digits()
CALLING: words()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'forty two'
```

```
>>> int(words()) + int(digits())
CALLING: words()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'forty two'
```

Built-In Exceptions

Most errors you see in Python are exceptions:

- `TypeError` for incompatible types
- `ValueError` for bad values

```
>>> int([1.0, 2.0, 3.0])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: int() argument must be a string, a bytes-like object or a number,
not 'list'

>>> int("forty two")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'forty two'
```

Built-In Exceptions

- `KeyError` for dictionaries
- `IndexError` for lists

```
>>> zoo_animals = ["penguin", "otter", "giraffe"]
>>> print(zoo_animals[10])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
>>> atomic_numbers_nobles = {'He': 2, 'Ne': 10,
...   'Ar': 18, 'Kr': 36, 'Xe': 54}
>>> print(atomic_numbers_nobles['Rn'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Rn'
```

Built-In Exceptions

Essentially every error in Python is an exception.

- `SyntaxError` for invalid Python
- `NameError` for an unknown identifier
- 'MemoryError` for using too much RAM
- `IndentationError` for inconsistent indentation

ALL of these are exceptions!

When you use exceptions, you're building on something intrinsic to Python itself.

Flow Control

An exception often means an error. But it doesn't have to.

```
# Use "speedyjson" if available.
# If not fall back to "json" in the standard library.

try:
    from speedyjson import load
except ImportError:
    from json import load
```

The `load` function called in code will be the best available version.

KeyError

Another example:

```
try:
    atomic_number = atomic_numbers_nobles[symbol]
except KeyError:
    logging.warning('Cannot look up atomic number for symbol: %s', symbol)
    atomic_number = None
```

In single-threaded code, equivalent to this:

```
if symbol in atomic_numbers_nobles:
    atomic_number = atomic_numbers_nobles[symbol]
else:
    logging.warning('Cannot look up atomic number for symbol: %s', symbol)
    atomic_number = None
```

Multiple Except Blocks

```python
try:
    value = int(user_input)
except ValueError:
    print("Bad value from user")
except TypeError:
    print("Invalid type (probably a bug)")
```

Often useful with logging:

```python
try:
    value = int(user_input)
except ValueError:
    logging.error("Bad value from user: %r", user_input)
except TypeError:
    logging.critical(
        "Invalid type (probably a bug): %r", user_input)
```

Cloud Computing

```python
# fleet_config is an object with the details of what
# virtual machines to start, and how to connect them.
fleet = CloudVMFleet(fleet_config)
# job_config details what kind of batch calculation to run.
job = BatchJob(job_config)
# .start() makes the API calls to rent the instances,
# blocking until they are ready to accept jobs.
fleet.start()
# Now submit the job. It returns a RunningJob handle.
running_job = fleet.submit_job(job)
# Wait for it to finish.
running_job.wait()
# And now release the fleet of VM instances, so we
# don't have to keep paying for them.
fleet.terminate()
```

Cloud Computing

```
# fleet_config is an object with the details of what
# virtual machines to start, and how to connect them.
fleet = CloudVMFleet(fleet_config)
# job_config details what kind of batch calculation to run.
job = BatchJob(job_config)
# .start() makes the API calls to rent the instances,
# blocking until they are ready to accept jobs.
fleet.start()
# Now submit the job. It returns a RunningJob handle.
running_job = fleet.submit_job(job)
# Wait for it to finish.
running_job.wait()
# And now release the fleet of VM instances, so we
# don't have to keep paying for them.
fleet.terminate()
```

Imagine `running_job.wait()` raises a network-timeout exception. Now `fleet.terminate()` is never called.

Whoops. Expen$ive.

Finally! A Solution

```python
try:
    do_something()
except SomeException:
    handle_some_exception()
finally:
    code_that_runs_no_matter_what()
```

Save Your Bank Account/Job

Protect against this with `finally`:

```python
# timeout is an exception type (even though it's lowercase)
from socket import timeout

fleet = CloudVMFleet(fleet_config)
job = BatchJob(job_config)
try:
    fleet.start()
    running_job = fleet.submit_job(job)
    running_job.wait()
except timeout:
    logging.error('Network timeout running job %d', running_job.id)
finally:
    fleet.terminate()
```

`finally` and `except`

Used when you have a block of code that must ALWAYS be executed, no matter what.

```python
conn = open_db_connection()
try:
    do_something(conn)
finally:
    conn.close()
```

You can also have one (or more) except clauses:

```python
conn = open_db_connection(CONFIG)
try:
    do_something(conn)
except ValueError:
    logging.error("Bad data read from DB")
finally:
    conn.close()
```

Exceptions Are Objects

An exception is an instance of an exception class.

Often your `except:` clause will just specify the class. But sometimes you need the actual exception object.

Catch with "as":

```
try:
    int("hello world")
except ValueError as exception_object:
    message = exception_object.args[0]
    print("EXCEPTION MESSAGE:", message)

# Prints out:
# EXCEPTION MESSAGE: invalid literal for int() with base 10: 'hello world'
```

Exception Object Info

Exception objects can have helpful info. The attributes vary, but it will almost always have an `args` attribute.

```python
# Maps symbols (like "He") to names ("Helium")
ELEMENTS = { "H": "Hydrogen","He": "Helium", "Li": "Lithium",
             "Be": "Beryllium", "B": "Boron", "C": "Carbon",
             # And so on for all the other elements.
}
def show_elements(symbols):
    for symbol in symbols:
        print(f"{symbol:2} {ELEMENTS[symbol]}")
try:
    show_elements(["Ne", "Be", "Li", "Xa"])
except KeyError as err:
    missing_element = err.args[0]
    print("No element for symbol: " + missing_element)
```

```
Ne Neon
Be Beryllium
Li Lithium
No element for symbol: Xa
```

Raising Exceptions

```python
def positive_int(value):
    # Converts string value into a positive integer.
    number = int(value)
    if number <= 0:
        raise ValueError("Non-positive value: " + str(value))
    return number
```

Focus on the `raise` line:

- `raise` takes an exception object
- You instantiate `ValueError` inline

```
>>> positive_int(-7.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in positive_int
ValueError: Non-positive value: -7.0
```

Pop Quiz

```python
def positive_int(value):
    # Converts string value into a positive integer.
    number = int(value)
    if number <= 0:
        raise ValueError("Non-positive value: " + str(value))
    return number
```

What does `positive_int("not a number")` do?

Pop Quiz

```
def positive_int(value):
    # Converts string value into a positive integer.
    number = int(value)
    if number <= 0:
        raise ValueError("Non-positive value: " + str(value))
    return number
```

```
>>> positive_int("not a number")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in positive_int
ValueError: invalid literal for int() with base 10: 'not a number'
```

Lab: Exceptions

Lab file: `exceptions.py`

- In `labs` folder
- When you are done, study the solution - compare to what you wrote.
- … and then optionally do `exceptions_extra.py`

Instructions: `LABS.txt` in courseware.