Exceptions

Austin Bingham
COFOUNDER - SIXTY NORTH

@austin_bingham

Robert Smallshire
COFOUNDER - SIXTY NORTH

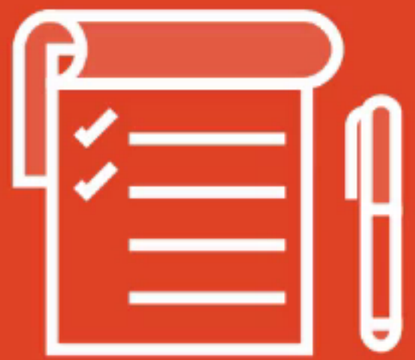
@robsmallshire

Exceptions

Read non-existent file

Exception is raised

Overview



Exception concept

Raising exceptions

Control flow

Catching exceptions

Unhandled exceptions

Use in Python

Built-in exceptions

Programmer vs. user errors

Resource cleanup

Exception handling

Mechanism for interrupting normal program flow and continuing in surrounding context

Exceptions: Key Concepts

1. **Raising** an exception
2. **Handling** an exception
3. **Unhandled** exceptions
4. Exception **objects**

Python exceptions are similar to exceptions in languages like C++ and Java

Exception Spectrum

Cleanup Actions

try...finally

```
try:
    # try-block
finally:
    # executed no matter how the
    # try-block terminates
```

Not Exception-safe

```
import os


def make_at(path, dir_name):
    original_path = os.getcwd()
    os.chdir(path)
    os.mkdir(dir_name)
    os.chdir(original_path)
```

Cleans up from Exceptions

```
import os
import sys


def make_at(path, dir_name):
    original_path = os.getcwd()
    os.chdir(path)
    try:
        os.mkdir(dir_name)
    finally:
        os.chdir(original_path)
```

Handle Exception and Cleanup

```
import os
import sys

def make_at(path, dir_name):
    original_path = os.getcwd()
    os.chdir(path)
    try:
        os.mkdir(dir_name)
    except OSError as e:
        print(e, file=sys.stderr)
        raise
    finally:
        os.chdir(original_path)
```

Moment of Zen

Errors should never pass silently, unless explicitly silenced

Errors are like bells
And if we make them silent
They are of no use

Platform-specific Code

Platform-specific Modules

Use the `msvcrt` module

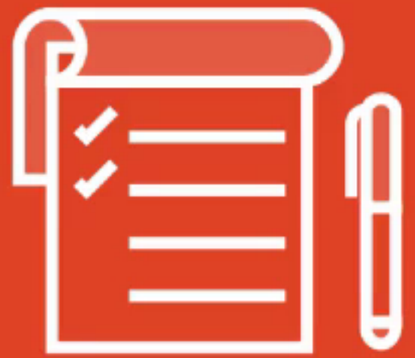
Use the `tty`, `termios`, and `sys` modules

```
        return msvcrt.getch()

except ImportError:

    import sys
    import tty
    import termios

    def getkey():
        """Wait for a keypress and return a single character string."""
        fd = sys.stdin.fileno()
        original_attributes = termios.tcgetattr(fd)
        try:
            tty.setraw(sys.stdin.fileno())
            ch = sys.stdin.read(1)
        finally:
            termios.tcsetattr(fd, termios.TCSADRAIN, original_attributes)
        return ch

    # If either of the Unix-specific tty or termios are not found,
    # we allow the ImportError to propagate from here
```

The caller can take alternative actions if both imports fail.

For example, they could downgrade to using `input()`.

Summary



Raising an exception interrupts program flow

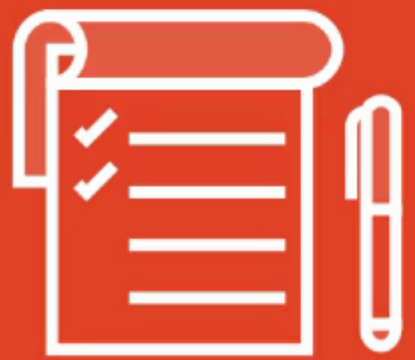
Handle exceptions with `try...except`

Exceptions can be detected within try-blocks

Except-blocks define handlers for exceptions

Python uses exceptions pervasively

Summary



Except-blocks can capture the exception

Avoid catching programmer errors

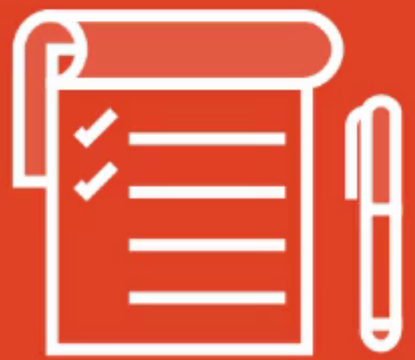
Signal exceptional conditions with `raise`

`raise` without an argument re-raises the current exception

Generally don't catch `TypeError`

Use `str()` to convert exceptions to strings

Summary

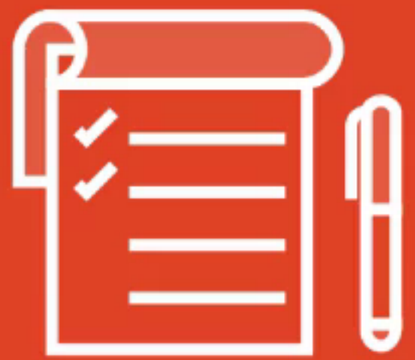


Exceptions are part of an API

Prefer built-in exception types when possible

Use `try...finally` for cleanup actions

Summary



`print()` output can be redirected

`!r` forces `repr` representations in f-strings

Python support logical `and` and `or` operators

Return codes are too easily ignored

Implement platform-specific actions with `ImportError` and EAFP

Exceptions and Control Flow

```
DIGIT_MAP = {
    'zero':  '0',
    'one':   '1',
    'two':   '2',
    'three': '3',
    'four':  '4',
    'five':  '5',
    'six':   '6',
    'seven': '7',
    'eight': '8',
    'nine':  '9',
}

def convert(s):
    number = ''
    for token in s:
        number += DIGIT_MAP[token]
    x = int(number)
    return x
```

◀ Filename: `exceptional.py`

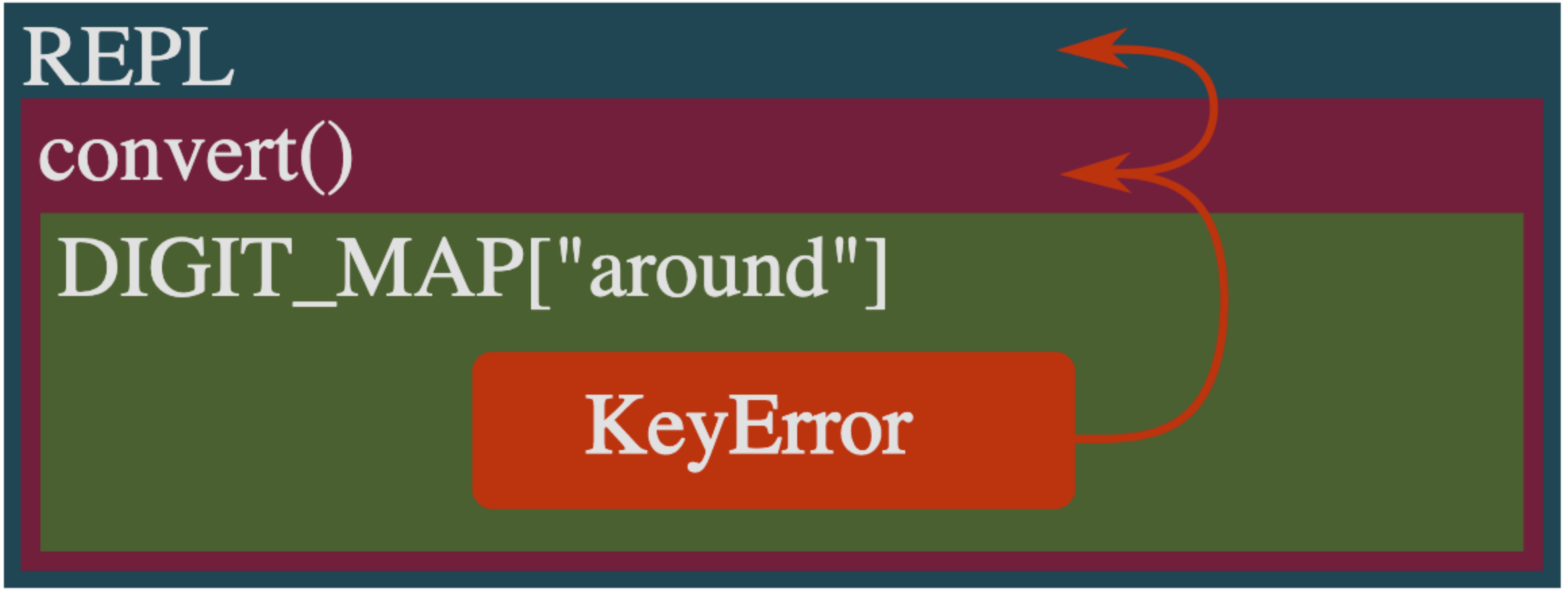
◀ Define a function

◀ Convert string to integer

◀ Return the integer

```
>>> from exceptional import convert
>>> convert("one three three seven".split())
1337
>>> convert("around two grillion".split())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/sixty-north/corepy/slide_spec/use-convert-v1/exceptional.py", lin
e 18, in convert
    number += DIGIT_MAP[token]
KeyError: 'around'
>>>
```

Exception Propagation



```
def convert(s):
    try:
        number = ''
        for token in s:
            number += DIGIT_MAP[token]
        x = int(number)
    except KeyError:
        x = -1
    return x
```

- try-block
- Raise exceptions

- except-block
- Handle exceptions

```
def convert(s):
    try:
        number = ''
        for token in s:
            number += DIGIT_MAP[token]
        x = int(number)
        print(f"Conversion succeeded! x = {x}")
    except KeyError:
        print("Conversion failed!")
        x = -1
    return x
```

◀ Print on success

◀ Print on failure

```
>>> from exceptional import convert
>>> convert("three four".split())
Conversion succeeded! x = 34
34
>>> convert("eleventeen".split())
Conversion failed!
-1
>>> convert(512)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/sixty-north/corepy/slide_spec/use-convert-v3/exceptional.py", lin
e 18, in convert
    for token in s:
TypeError: 'int' object is not iterable
>>>
```

```
def convert(s):
    try:
        number = ''
        for token in s:
            number += DIGIT_MAP[token]
        x = int(number)
        print(f"Conversion succeeded! x = {x}")
    except KeyError:
        print("Conversion failed!")
        x = -1
    return x
```

◀ Not executed

◀ Executed

```
def convert(s):
    """Convert a string to an integer."""
    try:
        number = ''
        for token in s:
            number += DIGIT_MAP[token]
        x = int(number)
        print(f"Conversion succeeded! x = {x}")
    except KeyError:
        print("Conversion failed!")
        x = -1
    except TypeError:
        print("Conversion failed!")
        x = -1
    return x
```

◀ Add `TypeError` handler

```
>>> from exceptional import convert
>>> convert(512)
Conversion failed!
-1
>>>
```

```
def convert(s):
    """Convert a string to an integer."""
    try:
        number = ''
        for token in s:
            number += DIGIT_MAP[token]
        x = int(number)
        print(f"Conversion succeeded! x = {x}")
    except KeyError:
        print("Conversion failed!")
        x = -1
    except TypeError:
        print("Conversion failed!")
        x = -1
    return x
```

- Duplication

- Add `TypeError` handler
- Duplication

```
def convert(s):
    """Convert a string to an integer."""
    x = -1
    try:
        number = ''
        for token in s:
            number += DIGIT_MAP[token]
        x = int(number)
        print(f"Conversion succeeded! x = {x}")
    except KeyError:
        print("Conversion failed!")
    except TypeError:
        print("Conversion failed!")
    return x
```

◀ Assignment

◀ Duplication

◀ Duplication

```
def convert(s):
    """Convert a string to an integer."""
    x = -1
    try:
        number = ''
        for token in s:
            number += DIGIT_MAP[token]
        x = int(number)
        print(f"Conversion succeeded! x = {x}")
    except (KeyError, TypeError):
        print("Conversion failed!")
    return x
```

◀ Merge except blocks

```
>>> from exceptional import convert
>>> convert("two nine".split())
Conversion succeeded! x = 29
29
>>> convert("elephant".split())
Conversion failed!
-1
>>> convert(451)
Conversion failed!
-1
>>>
```

```
def convert(s):
    """Convert a string to an integer."""
    x = -1
    try:
        number = ''
        for token in s:
            number += DIGIT_MAP[token]
        x = int(number)
    except (KeyError, TypeError):
    return x
```

```
>>> from exceptional import convert
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/sixty-north/corepy/slide_spec/use-convert-v7/exceptional.py", lin
e 24
    return x
         ^
IndentationError: expected an indented block
>>>
```

Exceptions resulting from programmer errors:

- `IndentationError`
- `SyntaxError`
- `NameError`

These should almost never be caught.

```
def convert(s):
    """Convert a string to an integer."""
    x = -1
    try:
        number = ''
        for token in s:
            number += DIGIT_MAP[token]
        x = int(number)
    except (KeyError, TypeError):
        pass
    return x
```

◀ pass is a no-op

```
def convert(s):
    """Convert a string to an integer."""
    try:
        number = ''
        for token in s:
            number += DIGIT_MAP[token]
        return int(number)
    except (KeyError, TypeError):
        return -1
```

Accessing Exception Objects

```
import sys

DIGIT_MAP =  . . .

def convert(s):
    try:
        number = ''
        for token in s:
            number += DIGIT_MAP[token]
        return int(number)
    except (KeyError, TypeError) as e:
        print(f"Conversion error: {e!r}",
              file=sys.stderr)
        return -1
```

```
>>> from exceptional import convert
>>> convert("fail".split())
Conversion error: KeyError('fail')
-1
>>>
```

```
# exceptional.py

from math import log

def string_log(s):
    v = convert(s)
    return log(v)
```

◀ Call `convert()`

◀ Compute natural log

Exceptions Can Not Be Ignored

Error codes are easy to ignore

Checks are always required

```
>>> from exceptional import string_log
>>> string_log("ouch!".split())
Conversion error: KeyError('ouch!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/private/var/folders/v6/1c70g62x2sb65cts_m6f7wdm0000gn/T/tmpjjna32c1/sli
de_spec/use-string-log/exceptional.py", line 32, in string_log
    return log(v)
ValueError: math domain error
>>>
```

```
def convert(s):
    try:
        number = ''
        for token in s:
            number += DIGIT_MAP[token]
        return int(number)
    except (KeyError, TypeError) as e:
        print(f"Conversion error: {e!r}",
              file=sys.stderr)
        raise
```

◀ Re-raise the exception

```
3.2188758248682006
>>> string_log("cat dog".split())
Conversion error: KeyError('cat')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/private/var/folders/v6/1c70g62x2sb65cts_m6f7wdm0000gn/T/tmpnhenuttf/slide_spec/use-convert-v11/exceptional.py", line 31, in string_log
    v = convert(s)
  File "/private/var/folders/v6/1c70g62x2sb65cts_m6f7wdm0000gn/T/tmpnhenuttf/slide_spec/use-convert-v11/exceptional.py", line 22, in convert
    number += DIGIT_MAP[token]
KeyError: 'cat'
>>> string_log(8675309)
Conversion error: TypeError("'int' object is not iterable")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/private/var/folders/v6/1c70g62x2sb65cts_m6f7wdm0000gn/T/tmpnhenuttf/slide_spec/use-convert-v11/exceptional.py", line 31, in string_log
    v = convert(s)
  File "/private/var/folders/v6/1c70g62x2sb65cts_m6f7wdm0000gn/T/tmpnhenuttf/slide_spec/use-convert-v11/exceptional.py", line 21, in convert
    for token in s:
TypeError: 'int' object is not iterable
>>>
```

Exceptions Are Part of the API

Heron of Alexandria's algorithm for calculating square roots

```
def sqrt(x):
    """Compute square roots using the method
    of Heron of Alexandria.

    Args:
        x: The number for which the square root
            is to be computed.

    Returns:
        The square root of x.
    """
    guess = x
    i = 0
    while guess * guess != x and i < 20:
        guess = (guess + x / guess) / 2.0
        i += 1
    return guess

def main():
    print(sqrt(9))
    print(sqrt(2))

if __name__ == '__main__':
    main()
```

◀ and tests of both conditions are true

```
$ python3 roots.py
3.0
1.4142135623730095
$
```

```
def sqrt(x):
    """Compute square roots using the method
    of Heron of Alexandria.

    Args:
        x: The number for which the square root
            is to be computed.

    Returns:
        The square root of x.
    """
    guess = x
    i = 0
    while guess * guess != x and i < 20:
        guess = (guess + x / guess) / 2.0
        i += 1
    return guess

def main():
    print(sqrt(9))
    print(sqrt(2))
    print(sqrt(-1))

if __name__ == '__main__':
    main()
```

◀ Take the square root of -1

```
$ python3 roots.py
3.0
1.414213562373095
Traceback (most recent call last):
  File "roots.py", line 25, in <module>
    main()
  File "roots.py", line 21, in main
    print(sqrt(-1))
  File "roots.py", line 13, in sqrt
    guess = (guess + x / guess) / 2.0
ZeroDivisionError: float division by zero
$
```

```
def sqrt(x):
    # . . .

def main():
    print(sqrt(9))
    print(sqrt(2))
    try:
        print(sqrt(-1))
    except ZeroDivisionError:
        print("Cannot compute square root "
              "of a negative number.")

    print("Program execution continues "
          "normally here.")


if __name__ == '__main__':
    main()
```

◀ try-block

◀ Catch ZeroDivisionError

```
$ python3 roots.py
3.0
1.414213562373095
Cannot compute square root of a negative number.
Program execution continues normally here.
$
```

```
def sqrt(x):
    # . . .

def main():
    try:
        print(sqrt(9))
        print(sqrt(2))
        print(sqrt(-1))
        print("This is never printed.")
    except ZeroDivisionError:
        print("Cannot compute square root "
              "of a negative number.")

    print("Program execution continues "
          "normally here.")


if __name__ == '__main__':
    main()
```

- try-block

- Never executed
- Catch `ZeroDivisionError`

Use Standard Exception Types

Standard types

Python provides standard exceptions types for signalling common errors

Invalid argument values

Use `ValueError` for arguments of the right type but with an invalid value

Exception constructors

Use `raise ValueError()` to raise a new `ValueError`

```
def sqrt(x):
    """Compute square roots using the method
    of Heron of Alexandria.

    Args:
        x: The number for which the square root
            is to be computed.

    Returns:
        The square root of x.
    """
    guess = x
    i = 0
    try:
        while guess * guess != x and i < 20:
            guess = (guess + x / guess) / 2.0
            i += 1
    except ZeroDivisionError:
        raise ValueError()
    return guess
```

◀ try-block

◀ Catch `ZeroDivisionError`

◀ Raise new exception

```python
def sqrt(x):
    """Compute square roots using the method
    of Heron of Alexandria.

    Args:
        x: The number for which the square root
            is to be computed.

    Returns:
        The square root of x.

    Raises:
          ValueError: If x is negative.
    """
    if x < 0:
        raise ValueError(
            "Cannot compute square root of "
            f"negative number {x}")

    guess = x
    i = 0
    while guess * guess != x and i < 20:
        guess = (guess + x / guess) / 2.0
        i += 1
    return guess
```

- document exceptions in docstring

- Test for negative argument
- Raise `ValueError`

```
$ python3 roots.py
3.0
1.414213562373095
Traceback (most recent call last):
  File "roots.py", line 41, in <module>
    main()
  File "roots.py", line 32, in main
    print(sqrt(-1))
  File "roots.py", line 17, in sqrt
    "Cannot compute square root of "
ValueError: Cannot compute square root of negative number -1
$
```

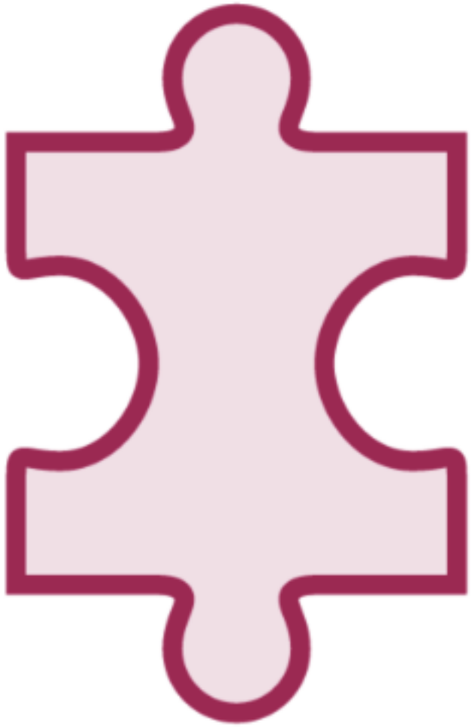
```python
import sys

def sqrt(): . . .

def main():
    try:
        print(sqrt(9))
        print(sqrt(2))
        print(sqrt(-1))
        print("This is never printed.")
    except ValueError as e:
        print(e, file=sys.stderr)

    print("Program execution continues normally here.")
```

- Catch `ValueError`
- Print the exception

```
$ python3 roots.py
3.0
1.414213562373095
Cannot compute square root of negative number -1
Program execution continues normally here.
$
```

Exceptions and Protocols

Exceptions and Protocols



Sequences should raise `IndexError` for out-of-bounds indexing.

Exceptions must be implemented and documented correctly.

Existing built-in exceptions are often the right ones to use.

Follow existing patterns

The more your code follows established patterns, the easier it will be for others to use.

Lookup Failure in Mappings

```
def lookup(key):
    if not find_key(key):
        raise KeyError()
    return value(key)
```

Common Exception Types

IndexError

An integer index is out of range

```
>>> z = [1, 4, 2]
>>> z[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
```

ValueError

An object is of the correct type but has an inappropriate value

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

KeyError

A lookup in a mapping failed

```
>>> codes = dict(gb=44, us=1, no=47, fr=33, es=34)
>>> codes['de']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'de'
>>>
```

Avoid Explicit Type Checks

```
def convert(s):
    if not isinstance(s, list):
        raise TypeError(
            "Argument must be a list")

    try:
        number = ''
        for token in s:
            number += DIGIT_MAP[token]
        return int(number)
    except (KeyError, TypeError) as e:
        print(f"Conversion error: {e!r}",
              file=sys.stderr)
        raise
```

- Check argument type
- Raise TypeError

```
def convert(s):
    # if not isinstance(s, list):
    #     raise TypeError(
    #         "Argument must be a list")

    try:
        number = ''
        for token in s:
            number += DIGIT_MAP[token]
        return int(number)
    except (KeyError, TypeError) as e:
        print(f"Conversion error: {e!r}",
              file=sys.stderr)
        raise
```

- Catch TypeError
- Re-raise it

Avoid Catching TypeError

Increase function reusability

Let TypeErrors arise on their own

It's Easier to Ask Forgiveness Than Permission

Prepare for Failure

Check all preconditions

Prepare for consequences

LBYL vs. EAFP

LBYL	EAFP
Look before you leap	Easier to ask forgiveness than permission

Python prefers EAFP

The code's "happy path" is emphasized rather than being interspersed with error handling

***Example:* file processing**

Processing details are not important

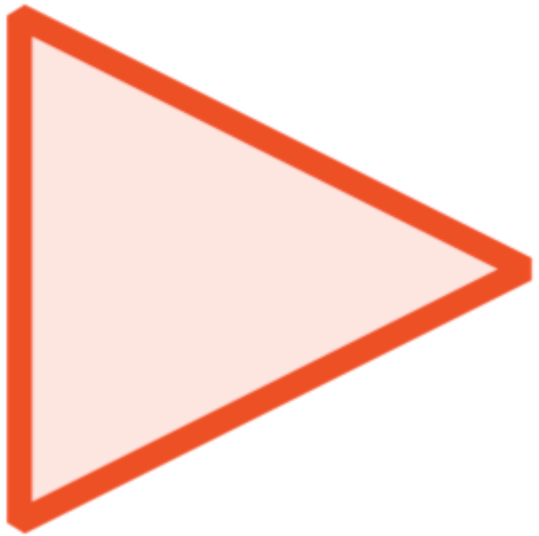
`process_file()` opens a file and reads it

```python
# Process file: LBYL

import os

p = '/path/to/datafile.dat'

if os.path.exists(p):
    process_file(p)
else:
    print(f'No such file as {p}')
```

- Check that the file exists
- File could be deleted after check

```python
# Process file: EAFP

p = '/path/to/datafile.dat'

try:
    process_file(f)
except OSError as e:
    print(f'Error: {e}')
```

◀ Handle `OSError`



EAFP is enabled by exceptions

Without exceptions, error handling is interspersed in program flow

Exceptions can be handled non-locally

EAFP plus Exceptions

1. Exceptions are **not easily ignored**
2. Error codes are **silent by default**
3. Exceptions plus EAFP makes it **hard for problems to be silently ignored**