INF1-IP

Introduction to Programming

Lecture 13 – Exception handling and floating-point values

Matteo Degiacomi

THE UNIVERSITY of EDINBURGH
informatics

> Programs meet the real world

Our code cannot stop these from happening, but it can be ready for them:

- A user types `twelve` where we asked for a number
- A settings file is missing
- A list that should hold values is empty

Today:

- Three kinds of problem: syntax, runtime, logic
- `try`, `except`, `else` and `finally`
- Raising our own exceptions
- Why `0.1 + 0.2 != 0.3`, and what to do about it



Three kinds of problem

```
if status = "open":
    print("Come in")
```

Syntax error: breaks Python's grammar, so no line runs. *Fix the code.*

```
→ SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?
```

```
quantity = int("hello")
```

Runtime exception: valid code, but an operation cannot complete. *Handle it, or fix the code.*

```
→ ValueError: invalid literal for int() with base 10: 'hello'
```

```
average = 4 + 8 / 2
print(average)
```

Logic error: runs, but gives the wrong answer. *Test, then fix the code.*

```
→ 8.0
```

Read the **last line** of a *traceback* first: the exception type, then its message.

Exceptions we have already met

Exception	Lecture	Example
`NameError`	L1	`print(morning)` before `morning` was assigned
`TypeError`	L2, L12	`"2" + 3`; a list as a dictionary key
`ValueError`	L2, L12	`int("4.5")`; unpacking the wrong number of items
`SyntaxError`	L3	`if status = "open":`
`IndentationError`	L3	indentation where Python does not expect it
`IndexError`	L5	a list position that does not exist
`KeyError`	L5	a dictionary key that does not exist
`AssertionError`	L9	a failed `assert`
`FileNotFoundError`	L10	opening a missing file for reading

Try it yourself #1: classify the problem

```
# A
for day in range(3)
    print(day)

# B
print("Total: " + 5)

# C: the percentage of 3 out of 4
percent = 3 / 4 * 10
print(percent)
```

- Is each one a **syntax error**, a **runtime exception** or a **logic error**?
- For A and B, which error does Python report?
- Fix each one

About 2 minutes

Example solution #1

```
# A: SyntaxError, the header needs a colon
for day in range(3):
    print(day)

# B: TypeError, convert the number first
print("Total: " + str(5))

# C: logic error, a percentage needs * 100
percent = 3 / 4 * 100
print(percent)
```

Output

```
0
1
2
Total: 5
75.0
```

Only B raises an exception while running.

Other fixes are possible, e.g., an f-string for B.

{ try and except

Predict: which messages are printed? And if `text` were `"12"`?

```
text = "twelve"
try:
    number = int(text)
except ValueError:
    print(repr(text), "is not a whole number")

print("The conversion attempt has finished")
```

```
'twelve' is not a whole number
The conversion attempt has finished
```

- `ValueError` in `try`: jump to the handler, **never back** to the failed line
- No exception: the handler is skipped
- Keep the `try` block **small**

Watch out: never use a bare `except:`. It also catches a typo such as `int(txet)`, a `NameError`, and reports it as bad input. *Name the exception you expect*.

Live coding: dividing a bill

Three friends type how many people share a **£60** invoice. What can go wrong?

```
answers = ["3", "two", "0"]

for answer in answers:
    people = int(answer)
    print(f"£{60 / people:.2f} each")
```

```
£20.00 each
ValueError: invalid literal for int()
with base 10: 'two'
```

- Then `"0"`: `ZeroDivisionError`
- One `except` clause per problem: the **first** match runs

Try it yourself #2: a safe ratio

Write `safe_ratio(numerator_text, denominator_text)`:

- Convert both input strings to floats and divide
- Catch `ValueError` and `ZeroDivisionError` separately
- Print a helpful message and return `None` for each

Try `("10", "4")`, `("ten", "4")` and `("10", "0")`.

About 3 minutes · in the notebook

Example solution #2

```
def safe_ratio(numerator_text, denominator_text):
    try:
        numerator = float(numerator_text)
        denominator = float(denominator_text)
        result = numerator / denominator
    except ValueError:
        print("Both values must be numbers")
        return None
    except ZeroDivisionError:
        print("The denominator cannot be zero")
        return None
    return result

print(safe_ratio("10", "4"))
print(safe_ratio("ten", "4"))
print(safe_ratio("10", "0"))
```

Output

```
2.5
Both values must be numbers
None
The denominator cannot be zero
None
```

{ else and finally

```
def demonstrate_sections(text):
    try:
        value = int(text)
    except ValueError:
        print("except: conversion failed")
        return None
    else:
        print(f"else: conversion produced {value}")
        return value
    finally:
        print("finally: attempt complete")
    print("after: is this ever printed?")

demonstrate_sections("25")
print("---")
demonstrate_sections("twenty-five")
```

- `else`: only if `try` succeeded
- `finally`: **always**, even before a `return`

Predict: what does each call print? Is the last line of the function ever printed?

```
else: conversion produced 25
finally: attempt complete
---
except: conversion failed
finally: attempt complete
```

Raising an exception

```
def mean(numbers):
    if len(numbers) == 0:
        raise ValueError("need at least one number")
    return sum(numbers) / len(numbers)
```

- `raise` stops the function
- The **caller** decides what to do

```
for numbers in [[2, 4, 6], []]:
    try:
        print(mean(numbers))
    except ValueError as error:
        print("No mean:", error)
```

```
4.0
No mean: need at least one number
```

Note: Not handled? The exception travels back to the notebook: **propagation**.
Use `raise`, not `assert`, for checks the program always needs.

Try it yourself #3: a percentage check

Write `checked_percentage(value)`:

- Return `value` if it is from 0 to 100
- Otherwise raise a `ValueError` with a message

Call it with `0`, `100`, `-1` and `110`, handling the errors in the calling code.

About 3 minutes · on your own

Example solution #3

```
def checked_percentage(value):
    if value < 0 or value > 100:
        raise ValueError("must be from 0 to 100")
    return value

for value in [0, 100, -1, 110]:
    try:
        print(checked_percentage(value))
    except ValueError as error:
        print(value, error)
```

Output

```
0
100
-1 must be from 0 to 100
110 must be from 0 to 100
```

`0` and `100` are **boundary** tests: both must be accepted.

Other solutions are possible.

An easy sum?

Predict: will the second line print `True`?

```
result = 0.1 + 0.2
print(result)
print(result == 0.3)
print(f"{result:.20f}")
```

```
0.30000000000000004
False
0.30000000000000004441
```

No exception: just a surprise.

- Floats are stored in **binary**, with a fixed number of digits
- In decimal, 1/3 = 0.333… never ends; in binary, 0.1 never ends
- Python stores the **nearest** float: close, not equal

Compare with math.isclose()

```
import math

print(math.isclose(0.1 + 0.2, 0.3))
print(math.isclose(1000.0, 1000.1, rel_tol=0.001))
```

```
True
True
```

Close if **either** tolerance is met.

Tolerance	Allowed difference	Default
`rel_tol`	a fraction of the larger value	`0.000000001`
`abs_tol`	a fixed amount, whatever the size	`0.0`

! Live coding: small errors add up

Code along:

- Add `0.1` ten times, and compare the total with `1.0`
- What about a result that should be zero, such as `0.1 + 0.2 - 0.3`?

Comparing with zero: with the default tolerances, `math.isclose(x, 0.0)` is `False` for any `x` that is not exactly zero.

- Give an `abs_tol` whenever the expected value is zero

> Summary

- Syntax errors: fix the code. Logic errors: test, then fix
- Runtime exceptions: `try` / `except SomeError:`, with a **specific** exception and a small `try`
- `else` runs on success; `finally` always runs
- `raise ValueError("...")` rejects a bad argument; the caller handles it
- Floats are approximations: compare with `math.isclose()`, not `==`

Homework: the consolidation challenge at the end of the notebook.