Programmieren I (Python)

Christian Osendorfer

2023-11-23

Errors, Exceptions, Testing, Debugging

Errors and Exceptions

- What happens when procedure executions hits an **unexpected condition** (not a **logical error/bug**!)?
- We get an **exception** to what was expected in the form of an error:
  - **SyntaxError**: Python can't parse program.
  - **IndexError**: Index in container does not exist.
  - **NameError**: local or global name not found.
  - **AttributeError**: attribute reference fails.
  - **TypeError**: operand doesn't have correct type.
  - **ValueError**: operand type okay, but value is illegal.
  - **IOError**: IO system reports malfunction (e.g. file not found).
  - **BaseException**: All errors are instances of **BaseException**! See here for more.

Dealing with exceptions

- Python can provide **handlers** for exceptions:

```
1 try:
2   a = 2
3   b = 0
4   print(a/b)
5 except:
6   print("Bug?!?")
```

- Exceptions **raised** by any statement in the body of `try` clause are **handled** by the `except` statement and execution continues with the body of the `except` statement.

Handling specific exceptions

- Have seperate except clauses to deal with a particalur type of exception

```
1  try:
2    a = int(input("Tell me a number"))
3    b = 0
4    print(a/b)
5  except ValueError:
6    print("Could not convert number")
7  except ZeroDivisionError:
8    print("Can't divide by zero")
9  except:
10   print("Something went wrong, if have no idea")
```

Other exceptions

- `else`: Body of this suite is executed when execution of associated `try` body completes with no exceptions
- `finally`: Body of this suite is **always executed** after `try`, `else` and `except`, even if they raise (see later) another error or executed a break, continue or return!
  - Useful for clean-up code that should be run no matter what else happened (e.g. close a file)

More Exceptions

```
1 try:
2   a = int(input("Tell me a number: "))
3   print(f'You told me {a}.')
4 except KeyboardInterrupt:
5   print("Woah, got interrupted")
```

Some Error subclasses have more attributes:

```
1 try:
2   a = int(input("Tell me a number: "))
3   print(f'You told me {a}.')
4 except ValueError as e:
5   print(e.args[0])
6   print(f'Woah, this is not a number!')
```

What to do with exceptions?

- Fail silently
  - just continue, maybe use some default values.
  - bad idea! User has no info the issue.
- Return an **error value**?
  - What value to choose?
  - Complicates code (e.g. caller needs to check for special values).
- Stop execution, **signal error** condition
  - In Python: **raise** an exception
  - `raise Exception("descriptive string")`

Raising an exception

```
1  def get_ratios(L1, L2):
2    """
3    Assumes: L1 and L2 are lists of equal length of numbers
4    Returns: a list containing L1[i]/L2[i]
5    """
6    ratios = []
7    for index in range(len(L1)):
8      try:
9        ratios.append(L1[index]/L2[index])
10     except ZeroDivisionError:
11       ratios.append(float('nan')) #nan = not a number
12     except:
13       raise ValueError('get_ratios called with bad arg')
14   return ratios
```

Assertions

- How can we ensure that **assumptions** on state of computation are as expected?
- The `assert` **statement** raises an `AssertionError` **exception** if assumptions are not met.
- Example of **defensive programming**.

Assertions

```
1 def avg(grades):
2   """
3   Compute the average of `grades`, a list of integers.
4   """
5   assert len(grades) != 0, 'No grades available!'
6   return sum(grades)/len(grades)
```

- Raises `AssertionError` if an empty list is passed in.
- Otherwise runs ok!
- Disable assertions! `python -O my_script.py`

Assertions

- We can also be less concise:

```
1 def avg(grades):
2   """ See before."""
3   if len(grades) == 0:
4     msg = "No grades available!"
5     err = AssertionError(msg)
6     raise err
7   return sum(grades)/len(grades)
```

Assertions as defensive programming

- Assertions don’t allow a programmer to control response to unexpected conditions.
- Ensure that execution halts whenever an expected condition is **not met**.
- Typically used to check inputs to functions, but can be used anywhere.
- Can be used to check outputs of a function to avoid propagating bad values.
- Can make it easier to locate a source of a bug.

When do we use assertions ?

- Goal is to spot bugs as soon as introduced and make clear where they happened.
- Use as a supplement to **testing**.
- Raise exceptions if users supplies bad data input.
- Use assertions to
  - check types of arguments or values.
  - check that invariants on data structures are met.
  - check constraints on return values.
  - check for violations of constraints on procedure (e.g. no duplicates in a list).
  - `pytest` !?

Types in Python!

- Providing type information can help catching a lot of simple bugs (and makes `assert` unnecessary for this category!)
- `def <function_name>(<parameter_name>: <parameter_type>) -> <return_type>:`
  - Called **type hints**/**type annotations** in Python.
- `def square(x: float) -> float:`
- The `typing` module has more fancy types! `List`, `Dict`, …
- `def avg(grades: List[int]) -> float:`
- Type hints have no runtime effect, they are not enforced on their own! Needs: A **static type checker**.
  - Let’s have a look at this!

Tests

- We know very well doctests: `python -m doctest script.py`
- Also possible:

```python
# some code with doctests

if __name__ == "__main__":
  import doctest
  doctest.testmod(verbose=True) # this is new!
```

- doctests are extremly useful. But the docstrings are usually simple cases meant to illustrate typical use cases.
- Complex doctests clutter code files.

Tests

- The module `pytest` allows to write tests in seperate files.
  - You need `pip install pytest`!!
- **Unit Test**: A block of code that checks for the correct behaviour of a function for one specific input.
- A **Test Suite** is a collection of tests that check the behavour of a function or a small set of functions. A test file contains a test suite.
- In `pytest` we
  - put a test suite in a file that starts with `test_`.
  - express a unit test with a function whose name starts with `test_`.
- The body of a unit test in `pytest` contains an `assert` statement.

Pytest

```python
# The file evens.py

def is_even(n:int) -> bool:
  """
  Return if `n` is even.

  >>> is_even(2)
  True
  >>> is_even(1)
  False
  """
  return n%2 == 0
```

Pytest

```python
# The file test_evens.py
import evens

def test_small_true() -> None:
  """Test on a small even number"""
  assert evens.is_even(2)

def test_large_true() -> None:
  assert evens.is_even(2**16)
```

Pytest

- How do we run `pytest`s?

```python
# At the bottom of test_evens.py
if __name__ == "__main__":
  import pytest
  pytest.main(['test_evens.py'])
```

- Or run `pytest` in root folder of project!
- Many (many!) options: `pytest test_evens.py::test_large_true`
  - See e.g. here for a lot of information about `pytest`.
- `pytest` will be run as part of a CI/CD workflow.
  - E.g. before every `git commit/push`.

Debugging

- Debug with `print`!
- Debug with `import pdb; pdb.set_trace()`
  - `pdb` is *built-in*, but there are many other options. E.g. `ipdb` (but needs `pip install`).
- Debug with `breakpoint()`
- Debug with tools from `vs-code`!