CS61A Lecture 26

Wednesday, October 30th, 2019

Announcements

- Guerilla section is on Saturday 11/2.

Pairs Review

In the 1950s, computer scientists liked to use confusing names:

- `cons` : Two-argument procedure that creates a pair
- `car` : returns the first element of a pair
- `cdr` : returns the second element of a pair (which is either `nil` or another pair)
- `nil` : the empty list

Scheme will use a dotted list to show that a list is not well-formed:

```
> (define x (cons 1 2))
> x
(1 . 2)
```

That's all the Scheme we're gonna do for today. Over the next couple of weeks, we will building an interpreter for Scheme using Python, but to do that, we're gonna need to learn more Python.

The only way to effectively do that is to learn exceptions in Python.

Exceptions

Computer programs can behave in ways we don't expect. Functions may receive an argument value of an improper type; some resource is not available etc.

The point of exceptions is to describe what to do in case of an error: to declare and respond to exceptional conditions, instead of stopping the program.

Python raises an exception whenever an error occurs.

Every error we've ever caused so far is an unahndled exception, which causes Python to print a stack trace.

Exceptions themelves are objects! They have classes with constructors. They enable non-local continuation of control (not like nonlocal statements, this is different!)

Let's use the following scenario: If `f` calls `g` and `g` calls `h`, there is an error in `h`, exceptions can shift control from `h` to `f` without waiting for `g` to return! (probably when `f` knows how to handle that error)

Exception handling tends to be slow, because it is non-standard.

Assert Statements

We've seen one method of exception handling before, `assert` .

Assert statements raise an exception of type `AssertionError`

```
assert <expression>, <string>
```

Assertions are designed to be used liberally. They can be ignored to increase efficiency of running Python with the "-O" flag: "O" stands for optimized.

```
$ python3
>>> assert print(2), "yikes"
2
AssertionError

$ python3 -O
>>> assert print(2), "yikes"

# Nothing happens; the entire statement is ignored
```

Whether assertions are enabled is governed by bool `__debug__` .

Raise Statements

Exceptions are raised with a raise statement.

```
raise <expression>
```

where `<expression>` must evaluate to a subclass of BaseException or an instance of one.

- `TypeError` – A function was passed the wrong number/type of argument
- `NameError` – A name wasn't found
- `KeyError` – A key wasn't found in a dictionary
- `RuntimeError` – Catch-all for troubles during implementation

Exceptions are constructed like any other object.

```
>>> TypeError("You passed in str instead of int")
TypeError: You passed in str instead of int
```

Try Statement

Try statements handle exceptions. They describe what happens if an exception is raised (by a raise statement!).

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
```

Execution rule for try

1. The `<try suite>` is executed first
2. If, during the course of executing the `<try suite>`, an exception is raised thaat is not handled otherwise, and:
3. If the class of the exception inherits from `<exception class>`,
4. The `<except suite>` is executed, with `<name>` bound to the exception.

The try statement is very smart. It can handle inheritance and parent functions:

In scenario 1, the try suite contains function `f`, which has a child function `g`, which has a child function `h`, and `h` runs into an error, the try suite will still handle the error.

In scenario 2, the exception class handles basic exceptions. If your exception class inherits from the base exception class, then it will still handle using that except suite.

Handling Exceptions

Exception handling prevents program from terminating:

```
>>> try:
...     x = 1/0
... except ZeroDivisionError as e:
...     print('handling a', type(e))
...     x = 0
...
handling a <class 'ZeroDivisionError'>
>>> x
0
```

> **Empty suites**
> You can specify a suite that does nothing when executed using the pass statement:
>
> ```
> >>> if 3 > 2:
> ...     pass
> ...
> ```
>
> You can also just execute a string, which will do nothing:
>
> ```
> >>> if 3 > 2:
> ...     "yeah looks right"
> ...
> ```

Multiple try statements: control jumps to the except suite of the most recent try statement that handles the type of exception.

Try statements don't handle the argument call. For example:

```
>>> invert_safe(1/0)
"ZeroDivisionError: division by zero"
```

Once exceptions are handled, the rest of the program won't know!

```
>>> try:
...     invert_safe(0)
... except ZeroDivisionError as e:
...     print("Hello")
...
"division by zero"
```

The `hello` never gets printed, because as far as the outer try statement cares, the exception was handled and the internal code is fine.

Example: Reduce

Write a function `reduce` that combines elements of `s` pairwise using `f` starting with the `initial`.

```
def reduce(f,s,initial):
    """
    >>> reduce(mul,[2,4,8],1)
    64
    """
    for x in s:
        initial = f(initial,x)
    return initial
```

This is easy enough, right? Well, what if we wanted to divide? What happens if it errors?

```
>>> from math import truediv
>>> reduce(truediv,[2,4,0],1024)
ZeroDivisionError
```

We can write a function that handles this error:

```
def divide_all(n,ds):
    try:
        return reduce(truediv, ds, n)
    except ZeroDivisionError as e:
        return 'oops zero'
```

The nice thing about this is that we are using abstraction. We are separating our concerns so that our code is modular, and more easily maintained. We have one part of the code that knows how to combine the elements, and another part that knows how to handle the error if something unexpected happens.