LECTURE 29

RECURSION

MCS 260 Fall 2021

Emily Dumas

REMINDERS

- Work on Project 3 ASAP. Do not delay!
- Project 3 autograder opens by Monday.
- Homework 10 posted; due Tuesday at 10am.

OOP LOOSE END: PROTOCOLS

We implemented the sequence protocol last time. There are others.

- Iterator — creates an iterable
- Mapping — creates a dict-like type

Still more can be found in the collections.abc module, which contains classes you can subclass when implementing the protocols.

RECURSION

A function in Python can call *itself*. This can be useful, for example if the result of the function at one argument is easy to obtain from the result at another argument.

This technique is called **recursion**. A function which uses it is a **recursive function**.

FACTORIAL

The classic example of recursion (being easiest to understand) is the computation of factorials:

$n! = n \times (n-1) \times (n-2) \times \cdots 2 \times 1$

e.g. $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Critical observation: $n! = n \times (n-1)!$

RECURSIVE FACTORIAL

Let's build a function `fact(n)` that uses $n! = n \times (n-1)!$ as the basis of its operation.

CALL STACK

Python keeps track of all the function calls that are underway in a stack. Items on the stack indicate where the call originated.

Calling a function *pushes* an item on the stack.

Returning *pops* an item form the stack.

There is a maximum allowed stack size. Exceeding it is a **stack overflow**.

COMPUTING FACT(3)

If push is `list.append` and pop is `list.pop`:

```
call_stack == [
        ]
```

Note "top" of stack is the last element.

COMPUTING FACT(3)

If push is `list.append` and pop is `list.pop`:

```
call_stack == [
            Called fact(3) on line 30
        ]
```

Note "top" of stack is the last element.

COMPUTING FACT(3)

If push is `list.append` and pop is `list.pop`:

```
call_stack == [
            Called fact(3) on line 30,
            Called fact(2) on line 18
        ]
```

Note "top" of stack is the last element.

COMPUTING FACT(3)

If push is `list.append` and pop is `list.pop`:

```
call_stack == [
            Called fact(3) on line 30,
            Called fact(2) on line 18,
            Called fact(1) on line 18
        ]
```

Note "top" of stack is the last element.

COMPUTING FACT(3)

If push is `list.append` and pop is `list.pop`:

```
call_stack == [
            Called fact(3) on line 30,
            Called fact(2) on line 18,
            Called fact(1) on line 18 # returns 1
        ]
```

Note "top" of stack is the last element.

COMPUTING FACT(3)

If push is `list.append` and pop is `list.pop`:

```
call_stack == [
            Called fact(3) on line 30,
            Called fact(2) on line 18
        ]
```

Note "top" of stack is the last element.

COMPUTING FACT(3)

If push is `list.append` and pop is `list.pop`:

```
call_stack == [
            Called fact(3) on line 30,
            Called fact(2) on line 18 # returns 2
        ]
```

Note "top" of stack is the last element.

COMPUTING FACT(3)

If push is `list.append` and pop is `list.pop`:

```
call_stack == [
            Called fact(3) on line 30
        ]
```

Note "top" of stack is the last element.

COMPUTING FACT(3)

If push is `list.append` and pop is `list.pop`:

```
call_stack == [
            Called fact(3) on line 30 # returns 6
        ]
```

Note "top" of stack is the last element.

COMPUTING FACT(3)

If push is `list.append` and pop is `list.pop`:

```
call_stack == [
        ]
```

Note "top" of stack is the last element.

RECURSIVE LISTDIR

How can we make a function `rlistdir(path)` that will return a list of the contents of a directory and all of its subdirectories?

Python actually has multiple functions in the standard library that can do this, though we haven't discussed them. The point is to construct a solution using the things we've covered!

RECURSION PROS AND CONS

Often can solve a problem with recursion or with loops (an **iterative** solution). Why use recursion?

Pros:

- Short code
- Clear code

Unclear:

- Speed

Cons:

- Uses more memory

REFERENCES

- In *Downey*:
  - Sections 5.8 to 5.10 discuss recursion

ACKNOWLEDGEMENTS

- Some of today's lecture was based on teaching materials developed for MCS 260 by Jan Verschelde.

REVISION HISTORY

- 2021-10-29 Initial publication