INF1-IP

Introduction to Programming

Lecture 18 – Recursion



THE UNIVERSITY of EDINBURGH
informatics



Let’s read some Dr Seuss!

> Recursion

Recursion means:

- Calling your function again and again (**step case**) – moving towards the base case
- Until you reach a given end (**base case**)

What do these look like for the Cat in the Hat?

- The **step case** is the current cat taking off its hat to reveal the next cat – each time getting closer to the
- … until the **base case** is reached: the cat that has Voom.

> Recursion

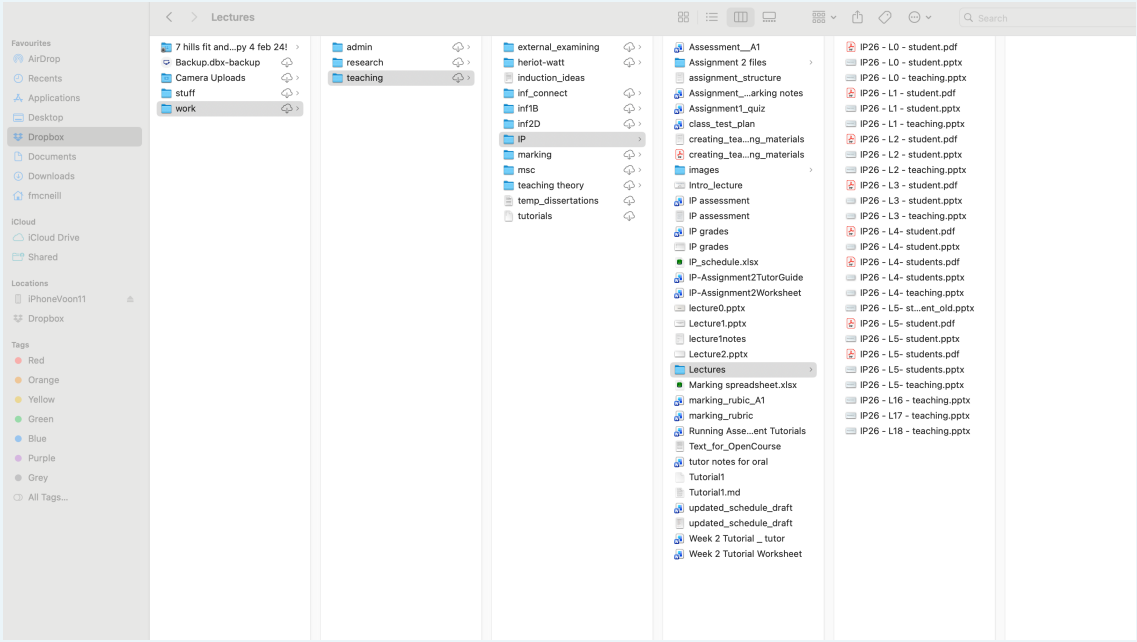
```
FUNCTION reveal_cat(cat):
    PRINT "Hello, Little Cat " + cat

    IF cat == "Z":
        PRINT "Release the Voom! Clean up the mess!"
        RETURN

    PRINT "Little Cat " + cat + " lifts his hat"
    reveal_cat(next_letter(cat))
```

> Another example – where are the slides?

- Step case: check my folder – if there are not there, go to the next level down
- Base case: found my folder!



Try yourself: pass the parcel

A children's game with a single present wrapped in layers and layers of wrapping paper. It is passed round a circle until the music stops, and then the child who has it takes off one layer – until the music stops.

What is the step case?
What is the base case?

Write some code that prints out 'not in layer [whatever layer you're at] – keep on passing!' until the present is finally found.

Try yourself: pass the parcel

```
def pass_the_parcel(layer):
    if layer == 0:
        print("I've won!!")
        return
    else:
        print("not in layer", layer, "- keep passing!")
        pass_the_parcel(layer - 1)


pass_the_parcel(5)
```

> How does recursion work?

- Recursion involves *a function calling itself.*
- If it does not reach the base case, it pauses the call and calls the next level down.
- After it reaches the base case it ‘unwinds’.
- This unwinding step is usually invisible – but it’s important to know it’s there so you can understand what’s happening.
- Let’s add a print statement to the code to make it explicit.

```
def pass_the_parcel(layer):
    if layer == 0:
        print("I've won!!")
        return
    else:
        print("not in layer", layer, "- keep passing!")
        pass_the_parcel(layer - 1)
        print("Finished", layer)


pass_the_parcel(5)
```

> How does recursion work?

```
pass_the_parcel(3): prints Not in layer 3, waits for pass_the_parcel(2)
  pass_the_parcel(2): prints Not in layer 2, waits for pass_the_parcel(1)
    pass_the_parcel(1): prints Not in layer 1, waits for pass_the_parcel(0)
      pass_the_parcel(0): prints I've won!!, returns
    pass_the_parcel(1): resumes, prints Finished 1, ends
  pass_the_parcel(2): resumes, prints Finished 2, ends
pass_the_parcel(3): resumes, prints Finished 3, ends
```

> How does recursion work?

- As you can see, `return` only ends the current call – it does not end the whole process, because there are still calls waiting.
- Sometimes after the base case, all we need to do is end each call – this is what happens with pass the parcel.
- But sometimes we do more work during the unwinding.

```
def pass_the_parcel(layer):
    if layer == 0:
        print("I've won!!")
        return
    else:
        print("not in layer", layer, "- keep passing!")
        pass_the_parcel(layer - 1)
        print("Finished", layer)


pass_the_parcel(5)
```

> Calculating factorials

- What is the value of n!
  - Is n-1 = 0? Base case
  - Multiple n by (n-1)! Step case

You go down to the base case before any calculations are done and then you build up.

Remember: 0! = 1

This is not something you can work out from the definition of factorial. It is something that is there to create a base case – otherwise factorial wouldn't work.

> Calculating factorials

```
def factorial(number):
    if number == 0:
        return 1

    result = number * factorial(number - 1)
    print("current result is", result)
    return result
```

> Calculating factorials

factorial(3): waits for factorial(2)
  factorial(2): waits for factorial(1)
    factorial(1): waits for factorial(0)
      factorial(0): base case, returns 1
    factorial(1): resumes, calculates 1 * 1 = 1, prints "Factorial of 1 is 1", returns 1
  factorial(2): resumes, calculates 2 * 1 = 2, prints "Factorial of 2 is 2", returns 2
factorial(3): resumes, calculates 3 * 2 = 6, prints "Factorial of 3 is 6", returns 6

print(factorial(3)): prints 6

> Calculating factorials

Let’s make our code more thorough.

It will work if we put in a positive whole number – but what if we don’t?

Let’s add some error checking to the code – as you saw in Lecture 13.

```
def factorial(number):
    if number < 0:
        raise ValueError("factorial is defined here for non-negative integers")

    if number == 0:
        return 1

    result = number * factorial(number - 1)
    print("current result is", result)
    return result
```

> Avoiding repetition

Sometimes recursion can lead to the same work being done twice.

In the Fibonacci series, n = n-1 + n-2

0, 1, 1, 2, 3, 5, 8, …

This means there are two recursion loops.

```
def fibonacci(index):
    if index < 0:
        raise ValueError("index must be non-negative")

    if index == 0:
        return 0

    if index == 1:
        return 1

    return fibonacci(index - 1) + fibonacci(index - 2)


print([fibonacci(index) for index in range(10)])
```

> Avoiding repetition

What calls are we making here?

```
fibonacci(4)
  fibonacci(3)
    fibonacci(2)
    fibonacci(1)
  fibonacci(2)     <- the same problem again
```

> Use iteration!

You can create the Fibonnacci series using recursion, but because of the repeated calls, this is inefficient. In this case, it is better to use *iteration.*

Recap: what is iteration?

Iteration means repeating a set of instructions, usually using a loop. We have seen this often throughout the course.

Rule of thumb: use a loop for “repeat this action”; consider recursion for “solve the same problem on a smaller part.”

Usually you want to use iteration. But in some situations, recursion is very efficient.

> Nested data

The Fibonnacci sequence is an example of a situation where we *can* use recursion, but it would be better to use iteration. Pass the parcel can easily be done with a loop.

So why would we ever want to use recursion?

The above two examples can easily use loops because *we know what the data looks like*. Recursion can be useful if we're not sure what to expect.

For example: we have list where items consist of nested lists. But we don't know how.

> Nested data

For example: I am very organised when I write my shopping lists. I like to categorise things in sublists.

```
shopping = [
    "bread",                                    # carbs
    ["milk", "cheese"],                         # dairy
    ["apples", ["carrots", "peas"]],            # fruit, veg
    ["rice", ["pasta", ["spaghetti", "penne"]]],   # rice, pasta
]
```

I may not be able to anticipate how deep each item will be. To make it easier to work with, I want to flatten the list so each item is just one deep. But how? Use recursion!

{ Let's code that

```
packing = [
    "laptop",
    ["charger", "adapter"],
    ["clothes", ["socks", "jumper"]],
]


def flatten(nested_items):
    result = []

    for item in nested_items:
        if isinstance(item, list):
            result.extend(flatten(item))
        else:
            result.append(item)

    return result


print(flatten(packing))
```

> When should we use recursion?

- If you want to keep solving the same problem on a subset of each item (think of the cats getting smaller)
- If you don't know how deep you need to get in each layer, which makes it difficult to create a loop
- If you don't need to go very deep
  - Every waiting call uses memory to store its local state and return position.
  - This is fine if you don't go very deep, but if you do you can risk running out of memory
  - In Python this isn't possible, because you get a `RecursionError` if you nest too deeply.

> Summary

- Recursion can be used in situations where:
  - You have a base case you are moving towards
  - And a step case them tells you how to move towards it
- It is especially useful if you don’t know the structure of the data you are working with
- It can be inefficient if used in inappropriate situations.