

Functional Programming in Python

RECURSION, ITERATORS, MAP, LAMBDA EXPRESSIONS, REDUCE,
FILTER, HIGHER ORDER FUNCTIONS

A solid orange horizontal bar spanning the width of the slide, located at the bottom.


Recursion

- A **recursive** solution to a problem features “self-similarity”, meaning that a function that solves a problem *calls itself*
- You actually already have familiarity with this concept
 - Consider the factorial operation in mathematics
 - $n! = n \times (n-1)!$ for integers $n \geq 1$, where $0! = 1$
 - Note how factorial is defined in terms of itself (i.e., the ! Symbol appears on both sides of the equals sign)
 - This is a **recursive definition** of factorial
 - The simplest case of a recursive definition is called the *base case*

Recursion Example: Factorial

- Writing a recursive Python function that implements factorial is very straightforward
- We need to define both the recursive part (which is when the factorial function calls itself), and the base case

```
def factorial(n):  
    if n == 0: Recursive call to factorial  
        return 1  
    else:  
        return n * factorial(n-1)
```



- See recursion_examples.py for code for many of the example recursive functions from these notes

Recursion

- All recursive functions have the following characteristics:
 - One or more **base cases** (the simplest cases) are used to stop recursion
 - One or more a **recursive calls** that reduce the original problem in size, bringing it increasingly closer to a base case until it becomes that case
 - A recursive call can result in many more recursive calls, because the method keeps on dividing a sub-problem into new sub-problems *that are of smaller size than the original*
 - These sub-problems are of the same nature as the original
- Please note: *solutions* can be recursive, not problems!

Trace: factorial(4)

`factorial(4) = 4*factorial(3)`
`factorial(3) = 3*factorial(2)`
`factorial(2) = 2*factorial(1)`
`factorial(1) = 1*factorial(0)`

} recursive function calls

`factorial(0) = 1`
`factorial(1) = 1*factorial(0) = 1*1 = 1`
`factorial(2) = 2*factorial(1) = 2*1 = 2`
`factorial(3) = 3*factorial(2) = 3*2 = 6`
`factorial(4) = 4*factorial(3) = 4*6 = 24`

} functions returning values

The diagram illustrates the recursive calls and return values for the factorial function. The first four lines show the recursive calls: factorial(4) calls factorial(3), which calls factorial(2), which calls factorial(1), which calls factorial(0). The next line shows factorial(0) returning 1. Then, factorial(1) is calculated as 1*1=1, factorial(2) as 2*1=2, factorial(3) as 3*2=6, and finally factorial(4) as 4*6=24. Arrows indicate the return values: a red arrow from 1 to factorial(0) in the next line, a blue arrow from 1 to factorial(1), a green arrow from 2 to factorial(2), and a purple arrow from 6 to factorial(3). The final result 24 is circled.

Example: Fibonacci Numbers

```
def fib(n):  
    if n == 0 or n == 1: # two base cases  
        return 1  
    return fib(n - 1) + fib(n - 2)
```

- Examples:

fib(0) = 1

fib(1) = 1

fib(2) = fib(1) + fib(0) = 1 + 1 = 2

fib(3) = fib(2) + fib(1) = 2 + 1 = 3

fib(4) = fib(3) + fib(2) = 3 + 2 = 5

Recursive Binary Search

- For recursive binary search (**rsearch**), the idea is basically the same as iterative binary search (But, the while-loop is replaced with a recursive call to the function)
- The algorithm checks the middle element to see if it equals the target
- If not, the function calls itself on the first half or second half, depending on whether the middle element is greater than or less than the target (respectively)

Completed rsearch Function

```
def rsearch(a, x, lower, upper):  
    if upper == lower + 1:  
        return None  
    mid = (lower + upper) // 2  
    if a[mid] == x:  
        return mid  
    if x < a[mid]:  
        return rsearch(a, x, lower, mid)  
    else:  
        return rsearch(a, x, mid, upper)
```


Base Conversion

Computer scientists and mathematicians often use numbering systems other than base 10. Write a program that allows a user to enter a number and a base and then prints out the digits of the number in the new base. Use a recursive function `baseConversion(num, base)` to print the digits.

Hint: Consider base 10. To get the rightmost digit of a base 10 number, simply look at the remainder after dividing by 10. For example, $153 \% 10$ is 3. To get the remaining digits, you repeat the process on 15, which is just $153 // 10$. This same process works for any base. The only problem is that we get the digits in reverse order (right to left).

The base case for the recursion occurs when `num` is less than `base` and the output is simply `num`. In the general case, the function (recursively) prints the digits of `num // base` and then prints `num % base`. You should put a space between successive outputs, since bases greater than 10 will print out with multi-character "digits." For example, `baseConversion (1234, 16)` should print 4 13 2.

Recursion exercises

Complete `recursion_exercises.py` and submit on blackboard.

Iterables

An iterable is anything you can iterate over.

- List

```
>>> for i in [1, 2, 3, 4]:  
...     print i,  
...  
1  
2
```

- Dictionary keys

```
>>> for k in {"x": 1, "y": 2}:  
...     print k  
...  
y  
x
```

Sets

```
>>> my_set = {1,2,3}  
>>> for x in my_set:  
...     print(x)  
...  
1  
2  
3
```

Iter function – Takes an iterable object and returns an iterator

```
>>> x = iter([1, 2, 3])  
>>> x  
<listiterator object at 0x1004ca850>  
>>> next(x)  
1  
>>> next(x)  
2
```

Map function

- The map function applies a function to every item in an iterable.
- Takes two inputs – function to apply and iterable object
- `map(function, iterable)`
- Squaring a number

```
x = [1, 2, 3, 4, 5]
def square(num):
    return num*num
print(list(map(square, x)))
```

Functional functions in Python are lazy.

If we didn't include the "`list()`" the function would store the definition of the iterable, not the list itself.

We need to explicitly tell Python "turn this into a list" for us to use this.

Example:

```
X = range(100)
```

```
X = list(range(100))
```

Lambda expressions

- Lambda expression is a one line function
- Lambda expression which cubes a given number

```
cube = lambda x: x * x * x
print(list(map(cube, x)))
```
- In a lambda expression, arguments go on the left hand side and functionality goes on the right hand side
- Simplifying square program in one line:

```
x = [1, 2, 3, 4, 5]
print(list(map(lambda num: num * num, x)))
```

Reduce function

- The reduce function turns iterable into one thing.
- `reduce(function, iterable)`
- Lambda expression can be used as the function
- Product of a list

```
from functools import reduce  
lst = [1,2,3,4,5,6,7,8]  
val = reduce((lambda x, y: x * y),lst)  
print(val)
```

Filter function

- The filter function takes an iterable and filters out all the things you don't want in that iterable.
- Normally filter takes a function and a list.
- It applies the function to each item in the list and if that function returns True, it does nothing.
- If it returns False, it removes that item from the list.
- E.g.

```
x = range(-5, 5)  
all_less_than_zero = list(filter(lambda num: num < 0, x))
```

Higher order functions

Higher order functions can take functions as parameters and return functions.

E.g.

```
def summation(nums):  
    return sum(nums)  
def action(func, numbers):  
    return func(numbers)  
print(action(summation, [1, 2, 3]))  
# Output is 6
```

Return functions

E.g.

```
def rtnBrandon():  
    return "brandon"  
def rtnJohn():  
    return "john"  
def rtnPerson():  
    age = int(input("What's your age?"))  
    if age == 21:  
        return rtnBrandon()  
    else:  
        return rtnJohn()
```


Questions?
