Lecture 5

Functions and Recursion

Burkay Genç, Ahmet Selman Bozkır, and Selma Dilek

29/03/2023

PREVIOUS LECTURE

- functions
- decomposition / abstraction
- scope

TODAY

- more functions
- recursion

Functions II

EXERCISE

- Write a function that computes the **least common multiple**: smallest number that is a multiple of two provided numbers.

EXERCISE

- To solve a problem
  - **understand** the problem
  - see **examples**
  - develop a **strategy**
  - **refine** the strategy
  - convert strategy to **code**,
  - **test** result
    - go back to previous steps if necessary (if something fails)

EXERCISE

- Write a function that computes the **least common multiple**: smallest number that is a multiple of two provided numbers.
- **understanding**
  - we are given two numbers: X and Y
  - we will find a number Z, such that
    - Z is perfectly divided by X
    - Z is perfectly divided by Y
    - Z is minimal (there is no smaller Z)

EXERCISE

- Write a function that computes the **least common multiple**: smallest number that is a multiple of two provided numbers.
- **examples**
  - For 2 and 3 -> 6
  - For 4 and 5 -> 20
  - For 4 and 6 -> 12
  - For 3 and 9 -> 9

EXERCISE

- Write a function that computes the **least common multiple**: smallest number that is a multiple of two provided numbers.
- **strategy**
  - let **BIG** be the larger number and **SMALL** be the smaller
  - if **BIG is divisible by SMALL**
    - then we are **done**, *return BIG*
  - otherwise, let **i = 2**
    - if **BIG * i** is divisible by SMALL
      - then we are **done**, return *BIG * i*
    - otherwise increment i and repeat the above check

EXERCISE

- Write a function that computes the **least common multiple**: smallest number that is a multiple of two provided numbers.
- **refine**
  - let **BIG** be the larger number and **SMALL** be the smaller
  - ~~if BIG is divisible by SMALL~~
    - ~~then we are done, return BIG~~
  - ~~otherwise~~, let **i = ~~2~~ 1**
  - ~~if~~ while **BIG * i** is *NOT* divisible by SMALL
    - ~~then we are done, return BIG * i~~
    - ~~otherwise~~ increment i ~~and repeat the above check~~
  - return **BIG * i**

EXERCISE

- Write a function that computes the **least common multiple**: smallest number that is a multiple of two provided numbers.
- **refine**
  - let **BIG** be the larger number and **SMALL** be the smaller
  - let **i = 1**
  - while **BIG * i** is *NOT* divisible by SMALL
    - increment i
  - return **BIG * i**

EXERCISE

- Write a function that computes the **least common multiple**: smallest number that is a multiple of two provided numbers.
- **code**

```
def LCM(X, Y):
   if X > Y:
      BIG, SMALL = X, Y
   else:
      BIG, SMALL = Y, X
   i = 1
   while not ((BIG * i) % SMALL == 0):
      i = i + 1
   return (BIG * i)
```

EXERCISE

- Write a function that computes the **least common multiple**: smallest number that is a multiple of two provided numbers.
- **test**

```
print(LCM(3, 5))
```

```
## 15
```

```
print(LCM(1, 4))
```

```
## 4
```

```
print(LCM(12, 4))
```

```
## 12
```

```
print(LCM(6, 8))
```

```
## 24
```

EXERCISE

- Write a program that sums up two rational numbers
- **understanding**
  - We are given two rational numbers: $\frac{a}{b}$ and $\frac{c}{d}$
  - We have to compute $\frac{a}{b} + \frac{c}{d}$
  - To do this, we need to find the LCM of b and d
  - Then we multiply a with $\frac{lcm}{b}$
    - and multiply c with $\frac{lcm}{d}$
  - When both denominators are equal to lcm, we sum up the numerators. The result is

$$\frac{a * lcm/b}{lcm} + \frac{c * lcm/d}{lcm} = \frac{a * lcm/b + c * lcm/d}{lcm}$$

EXERCISE

- Write a program that sums up two rational numbers
- **examples**
  - $\frac{2}{3} + \frac{4}{5} = \frac{22}{15}$
  - $\frac{1}{4} + \frac{1}{2} = \frac{3}{4}$
  - $\frac{3}{7} + \frac{5}{21} = \frac{14}{21}$
- Normally, $\frac{14}{21}$ should be simplified as $\frac{2}{3}$, but we will **omit** this for now.

EXERCISE

- Write a program that sums up two rational numbers
- **strategy**
  - ask for a, b, c and d
  - compute lcm = LCM(c, d) *reusing the LCM function from previous exercise*
  - output $\frac{a*lcm/b+c*lcm/d}{lcm}$

EXERCISE

- Write a program that sums up two rational numbers
- refine
  - There doesn’t seem to be any area for refinement

EXERCISE

- Write a program that sums up two rational numbers
- **code**

```
def sumRationals(a, b, c, d):
    lcm = LCM(b, d)
    numerator = int(a * lcm / b + c * lcm / d) #use int() to avoid x.0
    print(numerator, "/", lcm)
```

- **QUESTION**: why not return? why print?

EXERCISE

- Write a program that sums up two rational numbers
- **test**

```
# Normally this should be requested from the user using input()
sumRationals(2, 3, 1, 4)
```

```
## 11 / 12
```

```
sumRationals(1, 3, 1, 3)
```

```
## 2 / 3
```

```
sumRationals(4, 15, 2, 5)
```

```
## 10 / 15
```

RECURSION

Factorial

- Let $n!$ be the factorial of n,
  - $n! = 1 \times 2 \times 3 \times \ldots \times n-2 \times n-1 \times n$
  - $n! = (n-1)! \times n$
  - $(n-1)! = (n-2)! \times (n-1)$
  - $(n-2)! = (n-3)! \times (n-2)$
  - $(n-3)! = (n-4)! \times (n-3)$
  - ...
  - $1! = 1$

Factorial

- Let `fact(n)` be a function that computes factorial of n
- How do you implement it?

Factorial

- Let `fact(n)` be a function that computes factorial of n
- **strategy**
  - `fact(1) = 1` *this is the base case*
  - `fact(n) = fact(n - 1) * n` *this is the inductive step*
- `fact(n) = fact(n - 1) * n`
  - `fact(n - 1) = fact(n - 2) * (n - 1)`
  - `fact(n - 2) = fact(n - 3) * (n - 2)`
  - `fact(n - 3) = fact(n - 4) * (n - 3)`
  - …
  - `fact(1) = 1`

Factorial

- code

```
def fact(n):
   if n == 1:
      return (1)
   else:
      return (n * fact(n - 1))
```

- test

```
print(fact(2)) # 2x1
```

```
## 2
```

```
print(fact(5)) # 5x4x3x2x1
```

```
## 120
```

```
print(fact(7)) # 7x6x5x4x3x2x1
```

```
## 5040
```

Recursive Functions

- Recursion is a bit more difficult to think and code
- But if cleverly done, it results in much **shorter** and **cleaner** code

Fibonacci Numbers

- Can we implement Fibonacci numbers with recursion?
  - $F_n = F_{n-1} + F_{n-2}$
  - Yes!

```
def fibo(n):
   if n == 1 or n == 2:
      return (1)
   else:
      return fibo(n - 1) + fibo(n - 2)

print(fibo(5))
```

```
## 5
```

```
print(fibo(7))
```

```
## 13
```

Fibonacci Numbers

- What is really going on???
  - Let's change the code a bit to **debug** it

```
def fibo(n, depth):
   print(depth * "  " + "Computing fibo(" + str(n) + ")...")
   if n == 1 or n == 2:
      return (1)
   else:
      return fibo(n - 1, depth + 1) + fibo(n - 2, depth + 1)

print(fibo(5, 0))
```

```
## Computing fibo(5)...
##   Computing fibo(4)...
##     Computing fibo(3)...
##       Computing fibo(2)...
##       Computing fibo(1)...
##     Computing fibo(2)...
##   Computing fibo(3)...
##     Computing fibo(2)...
##     Computing fibo(1)...
## 5
```

String Reversal

- Can we reverse a string recursively?

String Reversal

- Can we reverse a string recursively?
  - Yes!

```
def reverse(s):
   if len(s) == 1:
      return (s)
   else:
      return ( s[-1] + reverse( s[:-1] ) )

print(reverse("burkay genc"))
```

```
## cneg yakrub
```

Palindromes

- Find whether a given string is a palindrome
  - palindrome: a string that is equal to its reverse

Palindromes

- Find whether a given string is a palindrome
- First solution with *reuse* of `reverse()`:

```
def isPalindrome(s):
   return s == reverse(s)

print(isPalindrome("burkay genc"))
```

```
## False
```

```
print(isPalindrome("abcdedcba"))
```

```
## True
```

Palindromes

- Find whether a given string is a palindrome
- Second solution with *recursion*:

```
def isPalindrome(s):
   if len(s) == 0 or len(s) == 1:
      return True
   else:
      return s[0] == s[-1] and isPalindrome(s[1:-1])

print(isPalindrome("burkay genc"))
```

```
## False
```

```
print(isPalindrome("abcdedcba"))
```

```
## True
```

IN CLASS EXERCISES

- Write a recursive function that checks whether a given sequence of numbers contains any negative numbers

IN CLASS EXERCISES

- Write a recursive function that checks whether a given sequence of numbers contains any negative numbers

```
def hasNegative(s):
    if len(s) == 0:
        return False
    return s[0] < 0 or hasNegative(s[1:])

print(hasNegative([1,2,3,4,5,6]))
```

```
## False
```

```
print(hasNegative([1,2,3,-4,5,6]))
```

```
## True
```

IN CLASS EXERCISES

DIFFICULT EXERCISE

Write a recursive function which takes two arguments: a sequence of numnbers and a target number; and returns True if a subset of the numbers in the sequence sum up to the target number.

- Example
  - Given [4,7,-2,7,5,5,-1] and a target value of 13, the return value is True, because 7 + 7 + (-1) = 13.
  - However, with the same sequence and a target value of 29, the return value is False.

IN CLASS EXERCISES

DIFFICULT EXERCISE

Write a recursive function which takes two arguments: a sequence of numnbers and a target number; and returns True if a subset of the numbers in the sequence sum up to the target number.

```
def subsetsum(s, t):
    if t == 0:          # if the target is 0, we are done
        return True
    elif len(s) == 0:   # and we know t != 0, because of the above condition
        return False
    else:               # we either have s[0] in the solution or not:
        return subsetsum(s[1:], t - s[0]) or subsetsum(s[1:], t)

print(subsetsum([4,7,-2,7,5,5,-1], 13))
```

```
## True
```

```
print(subsetsum([4,7,-2,7,5,5,-1], 29))
```

```
## False
```

HOMEWORK EXERCISES

1. Write a function that takes two strings S1 and S2, and returns the position index of S2 in S1. If S2 does not exist in S1, then the function returns -1.

2. Write a function `str_replace` that takes three strings, S1, S2 and S3; and replaces the first occurence of S2 in S1 with S3.

   - For example, `str_replace("burkay genc", "kay", "ak")` returns `"burak genc"`

3. Try to solve the first problem using a recursive strategy.

4. Using only 10 lira and 20 lira banknotes, in how many different ways can you pay 100 liras? Develop a recursive strategy to solve this problem.

   - The order of the banknotes are important. So `10+20+20+20+20+10` and `20+20+20+20+10+10` are counted as two different payments.

5. Now, try to convert the strategy you have developed in 4 to Python code. You have to create a function `pay(total, bn1, bn2)`. `total` is what you have to pay, `bn1` and `bn2` are the values of the banknotes you have to use. The function returns the number of ways this can be done.

Copyright Information