PYTHON EXERCISES

B. CSONKA, A. SIMON

1. Where to start?

- Where to find Python?
  - You can log in `tarski.math.bme.hu` the same way you log in to leibniz (if you are on leibniz, you just need to write `ssh -Y tarski` in a terminal) and you will find yourself in your home directory. There you can start `spyder3` or `ipython3` in a terminal. (There is `spyder` and `ipython` on leibniz, too, but avoid them because they are based on an old version of Python2.)
  - On your own windows machine you probably want this: `http://wiki.math.bme.hu/view/AnacondaInstall` With this, you'll get a graphical development environment called `spyder` that people seem to like.
  - `https://colab.research.google.com/` or `https://cocalc.com`. You need to register here, but get a nice a jupyter notebook.
  - `https://sagecell.sagemath.org/`, choose Python from the available languages. This is the worst choice.
- Lecture notes: `http://math.bme.hu/~asimon/info2/python.pdf`

2. Simple things

Exercise 2.1. Write code that prints in Celsius the temperature given in Fahrenheit stored in the variable `f`. The formula is $T_c = \frac{5}{9}(T_f - 32)$.

Exercise 2.2. Write code that prints all the numbers below the positive integer stored in the variable `n`. Do the same with the value of `n` included.

Exercise 2.3. Write code that prints the sum of the numbers not bigger than the positive integer stored in the variable `n`. Do the same with only the even numbers summed. (Do this without using `if`.)

Exercise 2.4. Write code that prints the factorial of the positive integer stored in the variable `n`.

Exercise 2.5. Write code that prints the (decimal) digits of the positive integer stored in `n` in reverse order. (Integer division is `//`)

Exercise 2.6. Write code that prints the sum of the (decimal) digits of the positive integer stored in `n`.

Exercise 2.7. Write code that prints the first `n` terms of the Fibonacci sequence. (It's not easy using only `while`. Hint: keep track of the last two terms.) Reminder: $F_0 = 0$, $F_1 = 1$, and $F_{n+2} = F_n + F_{n+1}$.

Date: January 30, 2025.

Exercise 2.8. Rewrite your solution to exercises 2.2-2.4 using `for` instead of `while`.

Exercise 2.9. Write a program that prints the sum of all the *positive* numbers in the list `l` of numbers. Use a `for` loop and do this with and without `continue`.

Exercise 2.10. Write a program using a `for` loop and `break` that prints the first proper divisor of the positive `int` stored in `n` if it has one, and does nothing otherwise.

Exercise 2.11. Write a function `compare()` of two arguments (both numbers), that prints `The numbers are equal.` if the two arguments are equal, `The first number is bigger.` if this is the case, and `The second number is bigger.` otherwise.

Exercise 2.12. Write a function `divisible()` of two arguments that returns `True` if its second argument is divisible by its first argument, and `False` otherwise.

Exercise 2.13. Write (using both the `if` statement and the `if` expression) a one-argument function `absval()` that returns the absolute value of the number given as its argument. (The built-in function `abs()` also does this.) Is `else` required in the version that uses the `if` statement?

Exercise 2.14. What does Python print after executing the following pieces of code?

```
x = 10

def f():
  x = 5

f()
print(x)
```

```
x = 10

def f():
  x = 5
  return x


x = f()
print(x)
```

Exercise 2.15. What are the values of the following expressions

- `s[3]`
- `s[-3]`
- `s[3:6]`
- `s[3:-4]`
- `s[4::2]`
- `s[-4::2]`
- `s[3::2]`
- `s[5::-2]`
- `s[5:1:-2]`

if `s = 'abcdefghij'`?

Exercise 2.16. Let `numbers = list(range(5))`. What's the difference between `numbers[1] = [True]` and `numbers[1:2] = [True]`? Is one of these equivalent to `numbers[1] = True`?

Exercise 2.17. Generate a list of the first 10 even numbers (a) using list comprehension (in two different ways) and (b) as a slice of a suitable `list`.

Exercise 2.18. Using nested list comprehensions, generate the following lists:

```
[[[0, 2], [0, 3], [0, 4]], [[1, 2], [1, 3], [1, 4]]]
```

and

```
[[[0, 2], [1, 2]], [[0, 3], [1, 3]], [[0, 4], [1, 4]]]
```

Exercise 2.19. Suppose that `a == 1`, `b == 2` and `c == 3`. Write *one* command that results in `a == 2`, `b == 3` és `c == 1`.

Exercise 2.20. Generate a list of the first 10 even numbers using a loop and the `.append()` method.

Exercise 2.21. Write a function `solve2()`, that, given the three coefficents of a second order equation, returns the list of its *real* roots. For example,

```
>>> solve2(1,1,-2), solve2(1,0,2)
([1.0, -2.0], [])
```

because $x^2 + x - 2 = (x - 1)(x + 2)$, and $x^2 + 2$ has no real roots. To compute square roots, use the `sqrt()` function of the `math` module.

Exercise 2.22. Write a function `sid(x,y)` (*s*afe *i*nteger *d*ivision) that for `int`s `x` and `y` returns their quotient as an `int` if `y` divides `x`, and the `float` `x/y` otherwise. For example,

```
>>> [sid(x,2) for x in [1, 4, 18530201888518410]]
[0.5, 2, 9265100944259205]
```

Exercise 2.23. Rewrite the following so it doesn't use `elif`:

```
>>> for i in [-5,5,15,25]:
...     if i<0:
...         print("negative")
...     elif i<=10:
...         print("small")
...     elif i<=20:
...         print("medium")
...     else:
...         print("big")
...
negative
small
medium
big
```

Exercise 2.24. Write a function `first_divisible(numbers,d)` that prints the first number in the list of numbers `numbers` that is divisible by d, or prints that there is no such number. For example,

```
>>> nums = [2, 5, 10, 14, 21, 35, 42, 51]
>>> first_divisible(nums, 7)
14 is divisible by 7

>>> first_divisible(nums, 13)
No number in the list is divisible by 13
```

Don't use `break` and `else`.

3. More challenging stuff

Exercise 3.1. Write a function of one argument, that, given a list of numbers $n_1, n_2, \dots, n_k$ returns a new list containing the numbers $n_1^3 - 1, n_2^3 - 1, \dots, n_k^3 - 1$. Do it both with and without using list comprehension.

Exercise 3.2. Write a function `squares()` of two arguments, so that `squares(m,n)` returns the list of squares between m and n. Do it both with and without using list comprehension.

Exercise 3.3. Generalize `squares()` of the previous exercise to every positive exponents! That is, using one of your implementations of `squares()` as your starting point, write `powers()`, so that `powers(m,n,k)` returns the list of the kth powers between m and n.

Exercise 3.4. Write a function `index_of()`, which, given an object and a list, returns the index of the first occurrence of the object in the list, or `None` if it doesn't occur in the list. (Recall that if a function doesn't do `return`, it returns `None`.)

Exercise 3.5. Write a function `indices_of()`, which, given an object and a list, returns the list of indices of the occurrences of the object in the list. For example:

```
>>> indices_of(3,[4,1,3,2,3])
[2, 4]
>>> indices_of(0,[4,1,3,2,3])
[]
```

Exercise 3.6. Write a function `substitute()`, which, given a list and two objects, returns a new list with all occurrences of the first object replaced by the second. For example:

```
>>> substitute([1,2,3,4,2],2,'a')
[1, 'a', 3, 4, 'a']
```

Exercise 3.7. Write a version of `substitute()` which doesn't return anything, but does the substitution on its list argument. For example:

```
>>> mylist = [1,2,3,4,2]
>>> substitute(mylist,2,'a')
>>> mylist
[1, 'a', 3, 4, 'a']
```

Exercise 3.8. Define a function `divisibles_by()` of two arguments, a list and an integer, which returns the list of those members of the list that are divisible by the integer. For example,

```
>>> divisibles_by(list(range(30,50)),7)
[35, 42, 49]
```

Exercise 3.9. Define a function `divisors()` that returns the list of proper divisors of its only argument.

Exercise 3.10. Define `sigma()` of one argument, which computes the number theoretic function $\sigma(n) = \sum_{1 \le d \le n, d|n} d$. Check it for a few values. For example:

```
>>> [(n,sigma(n)) for n in range (6,60,11)]
[(6, 12), (17, 18), (28, 56), (39, 56), (50, 93)]
```

Exercise 3.11. Define the function `is_perfect()` which returns `True` if its argument is a perfect number (which means that it is equal to the sum of its divisors smaller than itself) and `False` otherwise. Using your function, print all perfect numbers below 10000.

Exercise 3.12. Define a function `divisors_ival()` such that `divisors_ival(m,n)` prints the the list of proper divisors of all integers between m and n, like this:

```
>>> divisors_ival(30,35)
30 -> [2, 3, 5, 6, 10, 15]
31 -> []
32 -> [2, 4, 8, 16]
33 -> [3, 11]
34 -> [2, 17]
35 -> [5, 7]
```

Exercise 3.13. Write a function `is_prime` which returns `True` if its argument is a prime number and `False` otherwise.

Exercise 3.14. Write a function `primes_between()` such that `primes_between(m,n)` returns the list of primes in the interval $[m, n]$.

Exercise 3.15. Write a function `prime_divisors()` that returns the list of prime divisors of its argument.

Exercise 3.16. Write a function `max_exp()` such that `max_exp(m,n)` returns the biggest k such that $m^k \mid n$. You can assume $m > 1$.

Exercise 3.17. Write a function `prime_decomp()` that returns the prime decomposition of its argument as a list of pairs (or lists) whose first member is a prime divisor of the argument and the second is the exponent of the prime divisor in the prime decomposition. For example:

```
>>> prime_decomp(90)
[(2, 1), (3, 2), (5, 1)]
```

Exercise 3.18. Define a function `lookup()` of two arguments. The second is a list of 2-long tuples, and the first a "key". `lookup(key, list)` should return the second member of the first tuple in the list whose first member is equal to `key`, or `None` if there's no such member of the list.

Exercise 3.19. Using the functions in the two previous exercises, write a function `gcd()` that computes the greatest common divisor of its two arguments.

Exercise 3.20. Define a function that computes $\varphi(n)$ (the number of relative primes to n below n) for every natural number n.

Exercise 3.21. Write a function `separate()`, which, when called with a list l of numbers, returns a pair of lists, the first containing the negative, the second the non-negative members of l.

Exercise 3.22. Define a function `is_sorted()` of one argument, a list, which returns `True` if it is sorted in ascending order and `False` otherwise.

Hint: if you have trouble defining it, look up the definition of `repeats()` in the notes!

Exercise 3.23. Write a function which returns the minimal member of the nonempty list that is given as its only argument. Call it `my_min()`, because there is a built-in function `min()` which does this. Needless to say, don't use that. What other ways are there to cheat here? (It's good to be aware of the possibilities, but write a version without using any of them.)

Hint (for cheating): there's a built-in `max()` fuction.

Exercise 3.24. Write a function `min_index()` which returns the *index of* the first occurrence of the minimal member of the nonempty list of numbers that is given as its only argument.

Exercise 3.25. Write a function `min_indices()` which returns the list of the indices of the occurrences of the minimal member of the nonempty list of numbers that is given as its only argument. For example:

```
>>> min_indices([1,3,4,2,1,3,1,2])
[0, 4, 6]
```

Exercise 3.26. Write a function `nearest_to_avg()` that, given a list of numbers, returns the member closest to the average of the members (if it's not unique, then the one with the smallest index), or `None` if the list is empty. You can use the function `abs()`.

Exercise 3.27. Define a function `has_duplicates()` of one argument, a list, which returns `True` if there is an object which occurs at least twice in the list, and `False` otherwise.

Exercise 3.28. Write a function `longest_run()` of one argument, a list of numbers, which returns the length of the longest sequence of the same number. So, for example:

```
>>> longest_run([])
0
>>> longest_run([1])
1
>>> longest_run([1,2,3])
1
>>> longest_run([1,2,3,3,4])
2
>>> longest_run([1,2,2,2,1,2,3,3,4])
3
```

Exercise 3.29. Print the following ugly multiplication table:

```
1: 1 2 3 4 5 6 7 8 9
2: 2 4 6 8 10 12 14 16 18
3: 3 6 9 12 15 18 21 24 27
4: 4 8 12 16 20 24 28 32 36
5: 5 10 15 20 25 30 35 40 45
6: 6 12 18 24 30 36 42 48 54
7: 7 14 21 28 35 42 49 56 63
8: 8 16 24 32 40 48 56 64 72
9: 9 18 27 36 45 54 63 72 81
```

Hints: `print(42, end='whatever')` prints 42 followed by `whatever` instead of a newline. And `print()` simply prints a newline.

Exercise 3.30. Print all permutations (one by one) of 012!

```
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

Modify your program so that it prints the list of all such permutations as a list of list:

```
[[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]
```

Exercise 3.31. Write a function that computes the factorial of a natural number! Do a version without recursion, too. (C.f. 2.4, where we didn't have a choice!)

Exercise 3.32. Write a function `lucas(n)` which returns the nth Lucas-number. Lucas numbers are defined recursively as follows:

$$L_0 = 2, \qquad L_1 = 1, \qquad L_{n+2} = L_n + L_{n+1}$$

Try writing the function following the pattern of this definition, and another version that is not recursive.

Homework 3.1. Using the `lucas()` function (preferably the non-recursive one) from the previous exercise, print out the first few members of the subsequence of Lucas numbers whose indices are divisible by 3. Can you observe some regularity? If yes, can you verify it for the first 100 or so members of the subsequence? (Do this with the iterative version; if you only have the recursive one, make it 10 instead of 100.) Does this regularity hold if we consider indices whose remainders modulo 3 is 1 or 2? Can you formulate a theorem? If yes, can you verify it for the first few hundred Lucas numbers?

Exercise 3.33. Goldbach's conjecture says that every even number greater than 2 is the sum of two primes. Write a function `goldbach()`, which, given $1 < n \in \mathbb{N}$, returns the number of ways $2n$ can be written as the sum of two primes. ($p_1 + p_2$ and $p_2 + p_1$ count as the same decomposition.) For debugging your function, you may want to use `print()`, which, if given multiple arguments, will print them all, separated by a space.

Exercise 3.34. Convince yourself that Goldbach's conjecture will not be easy to refute, by exploring various intervals using your `goldbach()` function from the previous exercise.

For example: what's the minimal number of "Goldbach decompositions" of the 100 even numbers from 2000? 20000? 200000?

Exercise 3.35. Write a function `pascal()` that, given a positive natural number n, returns, as a list of lists, the first n lines of Pascal's triangle. Use the recursive definition of Pascal's triangle (this does not mean that your function must be recursive, see for example 3.31!): the nth row is of length n; the first row consists of one 1, and each entry of each subsequent row is the sum of the number above and to the left and the number above and to the right, treating blank entries as 0 (Wikipedia). For example:

```
>>> pascal(5)
[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]
```

You might want to use `n*[1]` to create a list of 1s of length n.

Exercise 3.36. An associative array is a data structure that maps values to keys. (Python has such a data structure: it's called `dict`, and we will use it later – but in this exercise we roll our own.)

There are four activites related to an associative array: we need to be able to create one, add key-value pairs to it, look up the value associated to a key in it, and remove a key-value pair from it.

We have already written the third component in Exercise 3.18. The first could simply be

```
def make_new_aa(): return list()
```

Write the remaining two: `insert(key,value,aa)` should return a copy of `aa`, either extended with the tuple `(key,value)` or, if `key` is already present in `aa`, it should return a copy of `aa` in which `(key,value')` is replaced by `(key,value)`.

Finally, `delete(key,aa)` should return a copy of `aa` with no tuple of the form `(key,value)`.

For example:

```
>>> d = make_new_aa()
>>> d = insert('one',1,d); d = insert('two',2,d); d = insert('three',3,d)
>>> d
[('one', 1), ('two', 2), ('three', 3)]
>>> lookup('two',d)
2
>>> d = insert('two',22,d); d = insert('four',4,d); d
[('one', 1), ('three', 3), ('two', 22), ('four', 4)]
>>> d = delete('five',d); d
[('one', 1), ('three', 3), ('two', 22), ('four', 4)]
>>> d = delete('two',d); d
[('one', 1), ('three', 3), ('four', 4)]
```

Exercise 3.37. Write a function `insert_sort()` of one argument, a list of numbers, which returns the sorted (in ascending order) version of the list.

You should use the Insertion sort algorithm, which works as follows: we read the elements of the list from left to right, and any time we encounter a member e that is smaller than the previous one, we shift to the right all the members that are left of e and are bigger than e, and insert e where the leftmost of these members were.

In other words, we scan the list from left to right, and whenever we find that the first k element is not in order, we correct this.

or

```
def insert_sort(lst):
    def insrt(x, l): #l is ordered
        if l == []: return [x]
        hd,*tl = l
        if x <= hd: return [x]+l
        return [hd] + insrt(x, tl)
    if lst == []: return lst
    return insrt(lst[0], insert_sort(lst[1:]))
```

4. I/O

Exercise 4.1. Write a program that asks for numbers, one after the other, and if the user presses RETURN without entering a number first, it returns the average of the numbers. So an interaction with your program should look something like this:

```
Enter a number: 1
Enter a number: 4
Enter a number: 12
Enter a number: 3
Enter a number:
The average is 5.0
```

Exercise 4.2. Write a program that asks for numbers separated by spaces, and prints their average. So an interaction with your program should look something like this:

```
>>> Enter some numbers separated by spaces: 1 4 13 2
The average is  5.0
>>>
```

Exercise 4.3. Write a function `read_first_lines()` of two arguments: the name of a file and an integer n. It should read and print the first n lines of the file (or all lines if there are no more than n lines in it).

Exercise 4.4. Write a function `copy_first_lines()` of three arguments: the name of an input and an output file, and an integer. It should do what the function in the previous exercise did, except that the output should go to the file named by the second argument. Check with an editor, or with `cat` or `less` at the command line that your function did what it's supposed to do.

Exercise 4.5. Write a function `count_lines()` of one argument, the name of a file. It should return the number of lines in the file (that's what `wc -l` does on the command line).

Exercise 4.6. Write a function `read_to_string()` that returns a string containing all the text in the text file named by its only argument. It shouldn't contain newline characters (\ns) but of course the last word of a line should not run into the first word of the next line.

Help: strings can be concatenated using `+`. And use `.rstrip()`, as in the lecture!

5. CONTAINERS

Exercise 5.1.* Why do you think the following works? Doesn't this contradict the fact that tuples are immutable?

```
>>> tup = ([0,1],[2]) ; tup[0][1] = 'a' ; tup
([0, 'a'], [2])
```

Hint: try it in Pythontutor!

Exercise 5.2. Write a function `list_diff()` of two arguments, both lists, which returns the list of those members of the first list (in the original order) which are not in the second. For example:

```
>>> list_diff(list(range(10)),list(range(0,15,3)))
[1, 2, 4, 5, 7, 8]
```

Exercise 5.3. Print the following pattern:

```
1
22
333
4444
55555
666666
7777777
88888888
999999999
```

Exercise 5.4. Write a function `wave()` of two arguments, such that for example `wave(5,2)` prints the pattern

```
o
oo
ooo
oooo
ooooo
ooooo
oooo
ooo
oo
o
o
oo
ooo
oooo
ooooo
ooooo
oooo
ooo
oo
o
```

Note that the maximum "height" is 5 and there are two waves.

If it works, write `new_wave()`, which is similar to `wave()`, except that there is only one row of maximum "height". For example, `new_wave(5,1)` should print

```
o
oo
ooo
oooo
ooooo
oooo
ooo
oo
o
```

Exercise 5.5. Write a function `merge()` which accepts two lists as arguments, and returns a new list that contains the first member of the first list, then the first member of the second list, then the second member of the first list, etc. In the first version, you can assume that the lists are of equal length. But in the final one if one of the lists is

longer, its remaining elements should come last, after the proper "merging" is done. For example:

```
>>> merge(list(range(5)),list(range(10,18)))
[0, 10, 1, 11, 2, 12, 3, 13, 4, 14, 15, 16, 17]
```

Exercise 5.6. Write a function `cat()` that prints in upper case the lines of the file whose name is given as its argument.

Exercise 5.7. Write a function that accepts one argument: the name of a text file. The file can be assumed to contain floating point numbers, one per line. It should return the average of these numbers.

*Hint:*Use the `float()` function to convert the string representation of a floating point number to a `float`. And `enumerate()` works for text files, too (see §2 of the lecture notes).

Exercise 5.8. Define a function `transpose()` of one argument, that, given a square matrix represented as a list of lists, returns its transpose. For example:

```
>>> transpose([[1,2,3],[4,5,6],[7,8,9]])
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Make a new list of lists, don't change the argument!

Exercise 5.9. Define a function `matrix_add()` of two arguments that returns the sum of two matrices (of the same shape) represented as lists of rows, where each row is represented as a list of numbers. For example:

```
>>> m1 = [[1,0,0],[0,1,0],[0,0,1]]
>>> m2 = [[1,2,3],[4,5,6],[7,8,9]]
>>> matrix_add(m1,m2)
[[2, 2, 3], [4, 6, 6], [7, 8, 10]]
```

Make a new list of lists, don't change either of the arguments!

Exercise 5.10. Define a function `matrix_mult()` of two arguments that returns the product of two matrices (of the appropriate shape) represented as lists of rows, where each row is represented as a list of numbers. For example:

```
>>> matrix_mult([[1,2],[3,4],[5,6]],[[1,0,0],[0,1,0]])
[[1, 2, 0], [3, 4, 0], [5, 6, 0]]
>>> matrix_mult([[1,0,0],[0,1,0]],[[1,2],[3,4],[5,6]])
[[1, 2], [3, 4]]
```

Make a new list of lists, don't change either of the arguments!

Exercise 5.11. Define a function `myzip2()` of two arguments, both sequences (lists or strings), that returns a list of tuples: the ith member of the jth tuple in the list should be the jth member of the ith argument. The length of the list should be the length of its shortest argument. For example:

```
>>> myzip2('abcdefg', list(range(4)))
[('a', 0), ('b', 1), ('c', 2), ('d', 3)]
```

Do *not* use `zip()`!

Exercise 5.12. Write a function `date_to_day()` that, given a month and a day in that month, returns the number of that day in the year. For example:

```
>>> date_to_day(3,15)     #31+28+15
74
```

We can assume that the year is not a leap year: so the lengths of the months are as in the following list:

```
MONTHS = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

Exercise 5.13. Write a function `day_to_date()` that is the inverse of `date_to_day()`. That is, given a number between 1 and 365, it returns the corresponding date (month, day pair). For example,

```
>>> day_to_date(74)
(3, 15)
```

As before, we can assume that the year is not a leap year and the lengths of the months are as in the following list:

```
MONTHS = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

Exercise 5.14. Write a function `lindex(string, substring)` that returns the smallest index in `string` where `substring` is found, or -1 if `substring` is not found. This is almost what the `.index()` method does, so don't use that! For example,

```
>>> lindex("At the turn of the century", "the")
3
>>> lindex("At the turn of the century", "them")
-1
```

Exercise 5.15. Write a function `count_occurrences()` of two arguments, both strings. It should return the number of times its second argument occurs as a substring in its first argument. For example:

```
>>> count_occurrences("At the turn of the century", "the")
2
>>> count_occurrences("At the turn of the century", "them")
0
```

Exercise 5.16. Suppose we have a text file `inventory.csv`, each line of which has the structure

good, *amount*, *unit price*

That is, within each line, the three "fields" are separated by commas. (This is a well known format called "comma separated values".) Write a short program that prints the content of each line in the format

`Name:` *good*
`Amount:` *amount*
`Unit price:` *unit price*

For example, if `inventory.csv` contains this:

```
ball,570,0.13
table,3,2000
racket,12,185
net,17,23
```

then the output should be

```
Good:  ball
Amount:  570
Unit price:  0.13
Good:  table
Amount:  3
Unit price:  2000
```

```
Good:  racket
Amount:  12
Unit price:  185
Good:  net
Amount:  17
Unit price:  23
```
Use an `f-string`!

Exercise 5.17. Modify your solution to the previous exercise so that it prints lines in the following format:

`Name:` *good*
`Total price:` *amount * unit price*

Don't forget to convert amounts and unit prices into numbers first!

Exercise 5.18. Using `f-strings`, print the following multiplication table:

```
    1  2  3  4  5  6  7  8  9
  +++++++++++++++++++++++++++
1:  1  2  3  4  5  6  7  8  9
2:  2  4  6  8 10 12 14 16 18
3:  3  6  9 12 15 18 21 24 27
4:  4  8 12 16 20 24 28 32 36
5:  5 10 15 20 25 30 35 40 45
6:  6 12 18 24 30 36 42 48 54
7:  7 14 21 28 35 42 49 56 63
8:  8 16 24 32 40 48 56 64 72
9:  9 18 27 36 45 54 63 72 81
```

Exercise 5.19. Suppose we have the mid-season standing of a football league as a csv file. The fields are: the name of the club, games played and points won by it. Write a function whose only argument is the name of the csv file, and which prints a nicely formatted list of the clubs ordered by the number of points *lost* (the less the better, of course) or the average points/game so far. (For the former, we need one additional information: each win is worth 3 points.) The idea is that these orderings give a better picture of how things stand when different clubs have not played the same number of matches. For example, if the csv file looks like this: `Man City,23,57`

```
Liverpool,22,48
Chelsea,24,47
Man Utd,22,38
West Ham,23,37
Arsenal,21,36
Tottenham,20,36
Wolverhampton,21,34
Brighton,22,30
Leicester,20,26
Aston Villa,21,26
Southampton,22,25
Crystal Palace,22,24
Brentford,23,23
Leeds,21,22
Everton,20,19
```

```
Norwich,22,16
Newcastle,21,15
Watford,20,14
Burnley,18,12
```

then the function should print this:

```
>>> by_lost_points("pl.csv")
   Club            GP Pts -Pts
 1 Man City        23 57  12
 2 Liverpool       22 48  18
 3 Tottenham       20 36  24
 4 Chelsea         24 47  25
 5 Arsenal         21 36  27
 6 Man Utd         22 38  28
 7 Wolverhampton   21 34  29
 8 West Ham        23 37  32
 9 Leicester       20 26  34
10 Brighton        22 30  36
11 Aston Villa     21 26  37
12 Southampton     22 25  41
13 Leeds           21 22  41
14 Everton         20 19  41
15 Crystal Palace  22 24  42
16 Burnley         18 12  42
17 Brentford       23 23  46
18 Watford         20 14  46
19 Newcastle       21 15  48
20 Norwich         22 16  50
```

Use `f-strings`!

Exercise 5.20. Define a function `is_palindrome()` that checks whether its argument, which is a sequence (a list, a tuple, or a string) is a palindrome, that is, it reads the same forwards and backwards. Don't reverse the sequence!

Exercise 5.21. Define a function `palindromes()` of one argument, the name of a text file (such as `/usr/share/dict/words`) with one word on each line, that prints all the palindromes in the file and returns the number of palindromes it has found.

5.1. `dict`.

Exercise 5.22. Define a function `substitute()` of two parameters: a string and a dictionary. It should return a new string which is a copy of the old one except that each character that is a key in the dictionary is replaced by the corresponding value (a string). For example:

```
>>> substitute("acbcade",{'a':'xyz','c':'zyx'})
'xyzzyxbzyxxyzde'
>>> substitute("acbcade",{'a':'c','c':'a'})
'cabacde'
```

Could this be done with repeated use of the `.replace()` method of the `str` class? Would it do the second example correctly?

Exercise 5.23. Modify your `substitute()` function of the previous exercise so that if the value associated in the dictionary to a character is `None`, it is deleted (replaced by the empty string `''`). For example:

```
>>> substitute("acbcade",{'a':'c','c': None})
'cbcde'
```

Exercise 5.24. Write a function `to_dict()` of one argument, a list of pairs (tuples). It should return a dictionary that has the first members of the tuples as keys, and the second members as corresponding values. For example:

```
>>> to_dict([('one',1), ('two',2), ('three',3)])
{'one': 1, 'two': 2, 'three': 3}
```

Exercise 5.25. Write a function `word_count()`, which, given the name of a text file, returns the number of words in it. Turn it into a standalone program that can be started from the command line, like this:

```
python word_count.py text.txt
```

This should return the same number as

```
wc -w text.txt
```

Try it on a longer text, such as this.

Exercise 5.26. Write a function `top_freq()`, which, given the name of a file and a natural number n, returns the list of the n most frequent words in the file, with their frequencies. For more realistic results, make sure "this" and "This" are treated as being the same word. Test it on a small text file where you know the word frequencies, and then try it on a longer text, such as this, for which it should return the list

```
[('the', 12436), ('and', 8311), ('of', 7327), ('a', 4997), ('to', 4412)]
```

Hints: the dictionary methods `.items()` returns the contents of the dictionary as a list of tuples of key-value pairs. You can sort a list of such tuples on their second members with

```
sorted(list_of_tuples, key=lambda x : x[1])
```

or

```
sorted(list_of_tuples, key=lambda x : x[1],reverse=True)
```

if you want descending order.

Exercise 5.27. Using your solutions to 5.24 and 5.26, write a function `top_freq_dict()` which does exactly what `top_freq()` does, but returns the result as a dictionary. For example, for this file, it should return the dictionary

```
{'the': 12436, 'and': 8311, 'of': 7327, 'a': 4997, 'to': 4412}
```

Hint: if the definition of your function is longer than two lines, then there's probably some misunderstanding.

Exercise 5.28. There is a datatype `set` in Python, but let's make a new one. Represent a set by a `dict` whose keys are the members of the set and whose values are all `None`. For example, union could be defined on this representation of sets like this:

```
>>> def union(s1,s2):
...     return dict.fromkeys(list(s1.keys()) + list(s2.keys()))
...
>>> union({1: None, 2: None}, {1: None, 3: None})
{1: None, 2: None, 3: None}
```

Here `dict.fromkeys()` does the same as `s1.fromkeys()` or `s2.fromkeys()`, but since the result (a new dictionary whose keys come from the iterable that is given as its argument and all of whose values are `None`) has nothing to do with either `s1` or `s2`, we can write `dict.fromkeys()`. (That is, `.fromkeys()` is a so called *class method*, not an *instance method*.)

Define `intersection()`, `difference()`, and also `set_add()`, which adds an element (its second argument) to the set (its first), and `set_remove()`, which removes an element (its second argument) to the set (its first) if it's present, and leaves it alone otherwise. For example:

```
>>> s1 = empty_set() ; set_add(s1,1) ; set_add(s1,2) ; s1
{1: None, 2: None}
>>> s2 = empty_set() ; set_add(s2,2) ; set_add(s2,3) ; s2
{2: None, 3: None}
>>> 1 in union(s1,s2)
True
>>> 1 in intersection(s1,s2)
False
>>> 1 in difference(s1,s2)
True
```

Fortunately, the `in` operator works automatically as expected, because if d is a `dict`, then `k in d` returns `True` iff `k` is a key in `d`.

Exceptions.

Exercise 5.29. Redo Exercise 5.14, but this time using the built in `.index()` method. The challenge is that `string.index(substring)` raises a `ValueError` exception when `substring` is not found in `string`, but `lindex()` should return -1. Other exceptions should go through, and *not* be hidden by returning -1 (or any other value, for that matter). For example,

```
>>> lindex("At the turn of the century", "the")
3
>>> lindex("At the turn of the century", "them")
-1
>>> lindex("At the turn of the century", 42)    #should raise an exception
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in lindex
TypeError: must be str, not int
```

Exercise 5.30. Modify your solution to the previous exercise in such a way that if an exception other than `ValueError` is raised by `.index()`, the message "Unknown error" is printed before the error is passed on to the caller.

For example,

```
>>> lindex("At the turn of the century", "the")
3
>>> lindex("At the turn of the century", "them")
-1
>>> #So far everything is as before. But note the (only) difference
>>> #in the following output!
```

```
>>> lindex("At the turn of the century", 42)
Unknown error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in lindex
TypeError: must be str, not int
```

5.2. **List comprehension.**

Exercise 5.31. Write a function `even_odd()` that accepts two lists of integers. It should return a list with all even members of the first list followed by the odd members of the second list. Use list comprehension!

Exercise 5.32. Using list comprehension, write a function `same_mod()` that accepts two lists of natural numbers and a positive natural numbers as arguments. It should return a list of all pairs of numbers whose first component comes from the first list, second component from the second list, and such that their remainders modulo the third argument are equal. For example:

```
>>> same_mod(list(range(4)), list(range(2,8)),3)
[(0, 3), (0, 6), (1, 4), (1, 7), (2, 2), (2, 5), (3, 3), (3, 6)]
```

Exercise 5.33. Modify your solution to the previous exercise so that `same_mod()` returns only those pairs which not only satisfy the condition in the previous exercise but also the extra condition that the first component of each pair is not divisible by the third argument. For example:

```
>>> same_mod(list(range(4)), list(range(2,8)),3)
[(1, 4), (1, 7), (2, 2), (2, 5)]
```

Exercise 5.34. Write a function `squares()` of two arguments: a list of integers and a positive natural number. `squares(lst,n)` should return a list of lists: the *i*th list in this list should contain the squares of the first `n` numbers starting from the *i*th member of `lst`. For example:

```
>>> squares(list(range(1,8,3)),3)
[[1, 4, 9], [16, 25, 36], [49, 64, 81]]
```

Use list comprehension(s)!

Exercise 5.35. Write a function `concatenate()` using list comprehension, that concatenates the list of lists that it is passed. For example:

```
>>> concatenate([list(range(3)),list(range(3,6)),list(range(6,8))])
[0, 1, 2, 3, 4, 5, 6, 7]
```

Exercise 5.36. As in 5.9, define a function `matrix_add()` of two arguments that returns the sum of two matrices (of the same shape) represented as lists of rows, where each row is represented as a list of numbers. But this time do it with the help of list comprehension!

Hint: use `zip()`!

Exercise 5.37.* Define a function `merge()` which merges two sorted list (that is, returns a sorted list with members of both lists). For example,

```
>>> merge([2,7,9], [1,5,6])
[1, 2, 5, 6, 7, 9]
```

Hint: try to write a recursive function, and think about what possibilities are there for the arguments.

6. FUNCTIONS

Exercise 6.1. What is the value of `mylist` after running the following piece of code? Why?

```
def side_effect_or_not (l):
    l = [2*i for i in l]
    return l

mylist = [1,2,3] ; side_effect_or_not(mylist)
```

Exercise 6.2. What is the value of `mylist` after running the following piece of code? Why?

```
def side_effect_or_not (l):
    l.append(len(l))
    return l

mylist = [1,2,3] ; side_effect_or_not(mylist)
```

Exercise 6.3. Let's call for the duration of this and the next exercise an object a tree if either it's a number, or a list of trees. So `0`, `1` and `2` are trees, because they are numbers, and thus `[0, 1, 2]` is also a tree, because it's a list of trees. Consequently,

```
[0, 1, [0, 1, 2], 2]
```

being a list of trees, is also a tree. And so is

```
[0, [0, 1, 2], [0, [0, 1, 2], 1, [0, 1, 2], 2], 2]
```

for the same reason.

Write a function `sumtree()` which computes the sum of the numbers of a tree. For example, it should return 14 for the last tree above.

Exercise 6.4.* Write a function `flatten()` that, given a tree, returns the list of the numbers it encounters by traversing it in a depth-first manner. "Depth-first" means starting from the beginning of the list (if the tree is a list — if it's not, then it is a number and the return value should be the list which has this number as its only element) and that whenever we encounter a list, we process that list first. A few examples should make it clearer:

```
>>> flatten(1)
[1]
>>> flatten([1])
[1]
>>> flatten([1, 2])
[1, 2]
>>> flatten([1, 2, [3, 4]])
[1, 2, 3, 4]
>>> flatten([0, [0, 1, 2], [0, [0, 1, 2], 1, [0, 1, 2], 2], 2])
[0, 0, 1, 2, 0, 0, 1, 2, 1, 0, 1, 2, 2, 2]
>>> flatten([[0, 1, 2], -1, [-2, [3, 4], [5, 6, 7, 8],
...     [9, 10, 11]], [13, 14, 15], [16, 17]])
[0, 1, 2, -1, -2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17]
```

Exercise 6.5.* Write a function `sublists()` that returns the list of all sublists (in any order) of its argument. (Here `l1` is a sublist of `l2` if it's a subsequence of `l2` in the sense

you learned in calculus. That is, all members of `l1` occur in `l2`, and in the same order.) For example:

```
>>> sublists([])
[[]]
>>> sublists([1])
[[1], []]
>>> sublists([1,2])
[[1, 2], [1], [2], []]
>>> sublists([1,2,3])
[[1, 2, 3], [1, 2], [1, 3], [1], [2, 3], [2], [3], []]
```

Exercise 6.6. Redo 3.18 but this time around `lookup()` should accept a keyword only argument called `default`, whose value it should return if it cannot find the key. If it's not given, `lookup()` should, as before, return nothing in this situation.

Exercise 6.7. Modify `concatenate()` from Exercise 5.35 so that instead of a list of lists, it accepts any number of list as arguments. For example:

```
>>> concatenate(list(range(3)),list(range(3,6)),list(range(6,8)))
[0, 1, 2, 3, 4, 5, 6, 7]
```

Hint: apart from one character, the two definitions should be the same.

Exercise 6.8. Define a function `myzip()` of any number of arguments, all sequences (lists or strings), that returns a list of tuples: the ith member of the jth tuple in the list should be the jth member of the ith argument. The length of the list should be the length of its shortest argument. For example:

```
>>> myzip('abcdefg', list(range(3)), list(range(10,14)))
[('a', 0, 10), ('b', 1, 11), ('c', 2, 12)]
```

Do *not* use `zip()`!

Hint: start from Exercise 5.11!

Exercise 6.9. As in Exercise 5.8, define a function `transpose()` of one argument, that, given a (not necessarily) square matrix represented as a list of lists, returns its transpose, but this time computed with list comprehension. For example:

```
>>> transpose([[1,2],[3,4],[5,6]])
[[1, 3, 5], [2, 4, 6]]
```

Hint: use `*` and `zip()`!

Exercise 6.10.* Redo Exercise 5.10 but this time using only list comprehension.

For simplicity, you can use `transpose()`.

Exercise 6.11. Define a function `apply()` of two arguments, which returns the result of applying its first argument, which should be a one-argument function, to its second.

Exercise 6.12. Write a function `self_compose` of two arguments such that

```
self_compose(fun, n)
```

where `fun` is a function of one argument and `n` is a natural number, returns `fun` composed with itself `n` times. In particular, `self_compose(fun, 1)` should return `fun` itself, `self_compose(fun, 2)` the composition of `fun` with itself, etc. What should `self_compose(fun, 0)` return? If you don't know, just assume (and check with `assert`) that the second argument is always positive.

For example:

```
>>> (self_compose(lambda x: x+1,3))(0)
3
```

Exercise 6.13. Write a function `compose12()` of two arguments, so that if `f1` is a function of one argument and `f2` is a function of two arguments, then `compose12(f1,f2)` returns the two argument function `f1(f2(_,_))`. For example,

```
>>> compose12(lambda x: 2*x,lambda x,y: x+y)(2,3)
10
```

Exercise 6.14. Write a function `compose1n()` of two arguments, so that if `f1` is a function of one argument and `f2` is a function of n arguments, then `compose1n(f1,f2)` returns the n argument function `f1(f2(_,_, ... , _))`. For example,

```
>>> compose1n(lambda x: 2*x,lambda x,y: x+y)(2,3)
10
>>> compose1n(lambda x: 2*x,lambda x,y,z: z*(x+y))(2,3,4)
40
```