Information Processing 1 — Week 7 exercises

Complete each exercise on your own laptop before the end of the exercise class. When you finish an exercise, **immediately** ask an instructor to check your answer. Answers checked after the end of the class will have a 50% penalty.

Do not struggle for more than a few minutes with anything you cannot solve. If you are stuck or have any questions then **raise your hand** and an instructor will help you.

1 [1 point] Wrapper functions

Most programming languages express angles in radians.[1] Most humans prefer to think in degrees.[2] MATLAB has the trigonometry functions `sin`, `cos`, `tan`, etc., that work in radians. It also has convenient functions `sind`, `cosd`, `tand`, etc., which work in degrees.

Python has `math.sin`, `math.cos`, `math.tan`, etc., that work in radians. It has no similar functions that work in degrees.

A 'wrapper' function is one that makes a trivial change to its parameter(s) and then calls another function. (The name comes from an analogy with a thin covering material 'wrapping' something much more substantial.)

Write some wrapper functions `sind`, `cosd`, and `tand` that work in degrees. Verify that they produce:

```
print(sind(  0))  #  0.0
print(sind(270))  # -1.0
print(cosd( 60))  #  0.5
print(tand( 45))  #  1.0
```

▷▷ Ask an instructor to check and record your work **now** (before you run out of time).

2 [3 points] Understanding and debugging recursive functions

The following program calculates the factorial of a non-negative integer.

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

2.1 Visualising program behaviour

Modify the `factorial(n)` function so that it prints "`factorial` *n*" every time it is called, and then prints "`returning` *result*" just before executing the `return` statement. Test your additions by printing the value of `factorial(5)`:

[1] Do you know why? One radian is the angle swept through by the radius of a circle as the point touching the circumference travels a distance exactly equal to the radius.

[2] Do you know why? You can blame the ancient Sumerians, Egyptians, and Babylonians who watched the skies and concluded that the sun travelled in a circular path and took 360 days to return to the same position as the previous year. (They were only 5.25 days wrong, quite an achievement for the time.) Hence the circle was divided into 360 parts. The Greeks adopted the Babylonian system and so here we are today. 360 turns out to be convenient also because it has 24 factors, more than any other number less than 720.

```
print(factorial(5))   # 120
```

If you do not understand the output, ask an instructor to help you to understand it.

2.2 Protecting functions against bad values

The above definition of `factorial(n)` does not handle negative arguments or non-integer arguments. Try it on a few 'bad' values to see what happens. For example:

```
print(factorial(5))
print(factorial(5.5))
print(factorial(-1))
print(factorial("x"))
```

Modify the `factorial(n)` function so that it checks the argument for non-integers and for negative values and, if necessary, prints a message explaining what is wrong with the argument and then returns `None`.

Hints:

1. To check if a value `n` is an integer you can use: `isinstance(n, int)`
2. To print a message and return `None` if some condition is true, you can use an `if` statement, like this:

```
if some-bad-condition-exists :
    print("message explaining the problem")
    return None
```

Test your function on the same four 'bad' inputs as above. If you have any difficulty making it work reliably, ask an instructor for help.

2.3 Guarding against bad values only once

The implementation of `factorial(n)` is not very efficient because it tests whether the argument is a non-negative integer every time it is called, even when it is called recursively from within itself. Because of the way the function works, you only really need to check the argument value once. If the argument passes the 'bad value' checks once, the recursive calls to `factorial(n-1)` are guaranteed to have only 'good' argument values.

Modify your function so that it only checks once whether the initial argument is valid.

One way to do this is to split `factorial(n)` into two functions. Perform the argument check in `factorial(n)` and, if it is OK, call another function (maybe called `_factorial(n)`) that calculates the actual result, recursively, without performing any checks on its argument's validity. (Don't forget that the recursive calls inside `_factorial(n)` need to be to `_factorial(n)` and not to `factorial(n)`.)

Test your program on the same 'bad' inputs.

▷▷ Show your three versions of `factorial` to an instructor **now**, before you run out of time..

3 [2 points] Input data and sentinel values

A 'sentinel' value is one that is used to indicate something interesting about input data. For example, to indicate the end of input data your program could check for a sentinel value that is illegal and therefore cannot occur in the actual input data.

The following program reads integers from the user one by one and prints out their values doubled.

```
while True:
    s = input("n: ")
    n = int(s)
    print(n * 2)
```

3.1 A sentinel value to terminate data input

Modify the program so that the loop ends when the user enters the value "`stop`" instead of an integer. Test the program in the obvious way.

3.2 Calculating averages from input data with a sentinel

Modify the program so that it prints out the total of the integers entered so far and their average. The program should still stop as soon as the user enters "`stop`".

Hint: You will need two variables, defined outside the loop, to hold the total value seen so far and the count of how many integers have been entered (which is needed to calculate the average).

Test your program on the integers from 1 to 5, whose total should be 15 and average 3.

▷▷ Show your final program to an instructor **now**, before you run out of time..

4 [4 points] Finding square roots using Newton's method

Newton's method for finding the square root s of a number n can be described in pseudo-code as follows:

- Set s to a guess of the square root of n (for example, start with $s = n/2$).
- While $s * s$ does not equal n, repeat:
  - Set $s = (s + n/s)/2$, which is a better approximation to the square root of n.

4.1 A simple implementation of Newton's method

Write a function called `newton(n)` which performs the above algorithm and then returns the result that it calculates.

Test your function with:

```
print(newton(100))
```

4.2 A better implementation, guaranteed to terminate

Test your function on the number 2 instead:

```
print(newton(2))
```

If the function no longer stops it is because floating point numbers are not stored precisely by the computer. A better version of the `newton(n)` function would check whether $s*s-n$ is 'close enough' to 0 and stop when it is.

Modify your program to stop when $s*s-n$ is closer than `0.00001` to the correct answer.

Hint: $s*s$ might be larger or smaller than n, so $s*s-n$ might be positive or negative. To avoid problems that this might cause you can use the `abs(x)` function which returns the absolute value of `x` (which is always positive). Comparing this with `0.00001` is then easy.

Test your program on several numbers, including: `1 2 10 100`

4.3 Generalisation: specifying the required accuracy

Add a second parameter '`epsilon`' to your function which specifies the maximum error allowed: `newton(n, epsilon)`

Test your function again on the numbers `1 2 10` and `100` while trying different values for epsilon including 0.1, 0.001, and 0.00001.

4.4 Relative accuracy

To make the `newton(n, epsilon)` function work better for very large and very small numbers, interpret epsilon as a *fraction* of `n` that is the maximum allowed error. For example, if you are looking for the square root of 1000, calling `newton(1000, 0.001)` would allow a maximum error of $1000*0.001=1$. Test your function on several numbers (including 1, 2, 10, and 100) using an epsilon of 0.0000001 (one part in ten million).

▷▷ Ask an instructor to check and record your work **now** (before you run out of time).

5 Challenge

Create a table of results for square roots to see the accuracy of your `newton()` function. Each row in your table should show `n`, `newton(n)`, `math.sqrt(n)`, and the absolute difference between the `newton(n)` and `math.sqrt(n)` results. Using the smallest value of `epsilon` that reliably terminates, mine looks like this:

```
n  newton(n)           math.sqrt(n)         diff
-  ---------          ------------         ----
1  1.0                1.0                  0.0
2  1.414213562373095  1.4142135623730951   2.220446049250313e-16
3  1.7320508075688772 1.7320508075688772   0.0
```

```
4  2.0                 2.0                 0.0
5  2.23606797749979    2.23606797749979    0.0
6  2.4494897427831788  2.449489742783178   8.881784197001252e-16
7  2.6457513110645907  2.6457513110645907  0.0
8  2.82842712474619    2.8284271247461903  4.440892098500626e-16
9  3.0                 3.0                 0.0
```

Reduce your value of `epsilon` until your results look similar to mine.

Hints:

- You can use a loop starting with '`for n in range(1, 10):`' to set `n` automatically.
- One way to make the columns line up is to write a function `pad(s, n)` that converts `s` to a string and then adds spaces after it so that the result it returns is exactly `n` characters wide.

If you are really curious about Newton's method, also print how many times the loop repeats before the error drops below the threshold.