# Problem Set 1

Physics 305, Fall 2020
Due Sept 4, at 5:00 pm on D2L

Before you begin work, please see the handout "General Instructions for Submitting Problem Sets". Note that this assignment is due before 5.00pm on Sept 4.

Also, see the second page of this file for some helpful hints on getting input into your program.

**Problem 1.** **[5 Points]** Write a simple program which asks for an integer from the terminal and prints out that many periods on a single line followed by the newline character, and then stops. When run, your program should produce output like:

```
> python dots_1_1.py
enter an integer: 27
...........................
> python dots_1_1.py
enter an integer: 5
.....
```

(of course,  is the bash prompt on my computer, and `dots_1_1.py` is the name of my program!)

**Problem 2.** **[10 Points]** Write a program that takes an integer as an argument from the command line (see second page) and then writes out the prime factorization of that number as a list of numbers on one line. Have your program print the prime factor and then the multiplicity of that factor after it in braces if it occurs more than once. When run, your program should produce output like:

```
bacchus {196} ./primefactor_1_2.py 3057403387500
  2{2}  3{3}  5{5}  7{7}  11
```

Hint: You can check to see if a number $d$ divides $n$ with no remainder by using the *modulus* function and checking the remainder is zero: `n%d==0`

You can read more about formatting the output of Python **`print`** statements in the on-line Python tutorial at `docs.python.org/3/tutorial/inputoutput.html`

**Problem 3.** **[10 Points]** Use bisection to determine the root nearest $x = 0$ of the function

$$f(x) = tan(x) - 3$$

Compare this to the analytic solution. This function has an infinite number of roots, so you may wish to plot the function in the vicinity of the desired root to get a feel for the shape of the function and what would be good bracketing values.

After writing code to plot the function, you might find it instructive to animate the progress of the bisection routine, showing on the plot the upper and lower bounds as they evolve.

**Problem 4.** **[10 Points]** Find, using Newton's method, the value of $x$ for which the function

$$f(x) = x^4 - 7x + 10$$

has its *minimum* and determine the minimum value of the function. Compare the value you find numerically with the analytic result.

**Problem 5. [15 Points]** Determine *all* of the roots of the polynomial

$$f(x) = x^3 - 25x^2 + 165x - 275$$

using the combined Bisection/Newton-Raphson iteration given in the notes. Write a single function which the returns both the value of the polynomial $f(x)$ and its derivative $f'(x)$, and pass it as an argument to your newton root-finding function. Make your function check its inputs (*i.e.* check that the root is really bracketed by the input bounds) and do any other "bomb-proofing" you can think of.

Again, make a plot of the function first to get a feel for the shape of the curve to determine a good initial guess for the root. (Different values of the initial guess will of course cause the iteration to converge to different roots.)

Have the code which calls your root-finding function demonstrate that the result returned is in fact correct within the given tolerance.

As a check, insert the roots you find back into the function and see how close these results are to zero.

## Scripts

To date we have been executing python programs by using the line
`python myprog.py`
Another way to execute code is by exploiting a feature of the shell.

The first line of every shell script can contain the special characters `#!` followed by the name of an interpreter which is then given the remaining contents of the file. So, to make a python program file executable, first make the first line of the file containing the python main routine
`#!/usr/bin/env python`

Next, we need to tell the shell that this file can be executed. We do this by changing the *mode* of the file to add the executable property:
`chmod +x myprog.py`
I can then execute my python script by typing
`./myprog.py`
much more like a command than typing "python" explicitly every time.

## Getting Input from the Keyboard

Python provides the function
`input(string)`
which returns a character string with whatever line you typed in. To separate this into individual tokens separated by whitespace, one can use the `.split()` member function of the string object:

```python
s = input("enter something: ")
q = s.split()
print("there were {0:d} tokens separated by whitespace".format(len(q)))
for i in range(len(q)):
    print(i, q[i])
```

Don't forget that everything you entered is a character string, *even the numbers*. To get actual numbers, you need to convert them from the character strings:

```python
s = input("enter two integers followed by a float: ")
q = s.split()
if len(q) !=3:
    print("bad input")
    exit()

int1 = int(q[0])
int2 = int(q[1])
f1 = float(q[2])

print(int1, int2, f1)
```

## Accessing Command Line Arguments

Often it is useful to give your Python program some input right from the command line:

```python
#!/usr/bin/env python
import sys

if len(sys.argv) != 4:
   print("Usage: {0:s} <int> <int> <float>".format(sys.argv[0]) )
```

```
    exit()

int1 = int(sys.argv[1])
int2 = int(sys.argv[2])
f1 = float(sys.argv[3])

print(int1, int2, f1)
```

Note that you don't have to call `.split()` on `sys.argv`; it is already split into a list of separate tokens. You still have to convert character strings to numeric values where appropriate, however.

## Checking Input

It is generally wise, or at least helpful, to have your program check to be sure that the input given is what you expect – for example, if you want a positive integer, that what was entered was indeed a positive integer.

The simplest way to do this is to use Python's **`assert`** statement:
assert <conditional>
will cause the program to crash with an error message if `<conditional>` is not true. This is very handy to have your code check on assumptions you have made in your algorithm.

Having the program crash, however, is not always what is desired. Python has a more gentle way of handling errors. Suppose you have a character string `s` which you wish to convert to an integer, and suppose this was entered by the user of your program. It may or may not represent a valid integer. We can write

```
try:
    n = int(s)
except ValueError:
    print(``Whoops! ``, s, `` is not an integer!'')
    exit(1)

# here, n is assured to be a valid integer
```

Python "tries" executing the statement after **`try`**; if it succeeds, control passes on to the rest of your program. If it fails, then, if the error was a **`ValueError`**, it executes the code in the **`except`** clause. If neither of these happens, then the program crashes anyway! You can read more about exception handling in the on-line Python tutorial at `docs.python.org/3/tutorial/errors.html`.