If you are still having trouble getting started with Python, you could try doing simple exercises on one of these sites.

- https://github.com/zhiwehu/Python-programming-exercises
- https://www.w3resource.com/python-exercises/python-basic-exercises.php
- https://www.practicepython.org/
- https://pynative.com/python-basic-exercise-for-beginners/

## 4.3

The complexity of the question isn't always indicative of the complexity of the answer.
A point in case is question 4.3, the answer of which is simple.
We need two functions, ALU and tick.
ALU takes two numbers A and B and a function (id) F and applies the indicated function to A and B, returning the result. SO, for instance, ALU(5,6,2) should return 30, because 2 signifies MULT.
ALU is therefore straightforward:

```
def ALU(A,B,F):
      if F==1:
            return A+B
      if F==2:
            return A*B
      if F==3:
            return A-B
      if F==4:
            return A^B
      if F==5:
            return ~A
      if F==6:
            return A//B
```

For now, we don't do flags.

Function 'tick' is even simpler. In the context of a list of registers R and given indices a, b and of registers and a function index f, 'tick' should call ALU and store the result in register c.

```
def tick(a,b,c,f):
      R[c] = ALU(R[a],R[b],f)
```

That's it. We're done. We can now execute the program described in the exercise.

## 4.2

Analysing the input is straightforward. Use either Python 'in' or the method 'startswith' to check what the input is. Then apply the appropriate conversion. I will write a function that converts from any radix based on the digits you give. This means that processing the input can be written as:

```
if inp.startswith("0b"):
      res = fromRdx(inp[2:],bits) #skip the 0b
elif inp.startswith("0x"):
      res = fromRdx(inp[2:],heds) #skip the 0x
else:
      res = fromRdx(inp,decs)
```

This assumes the digits for three bases are:

```
bits = ['0','1']
decs = ['0','1','2','3','4','5','6','7','8','9']
heds = ['0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F']
```

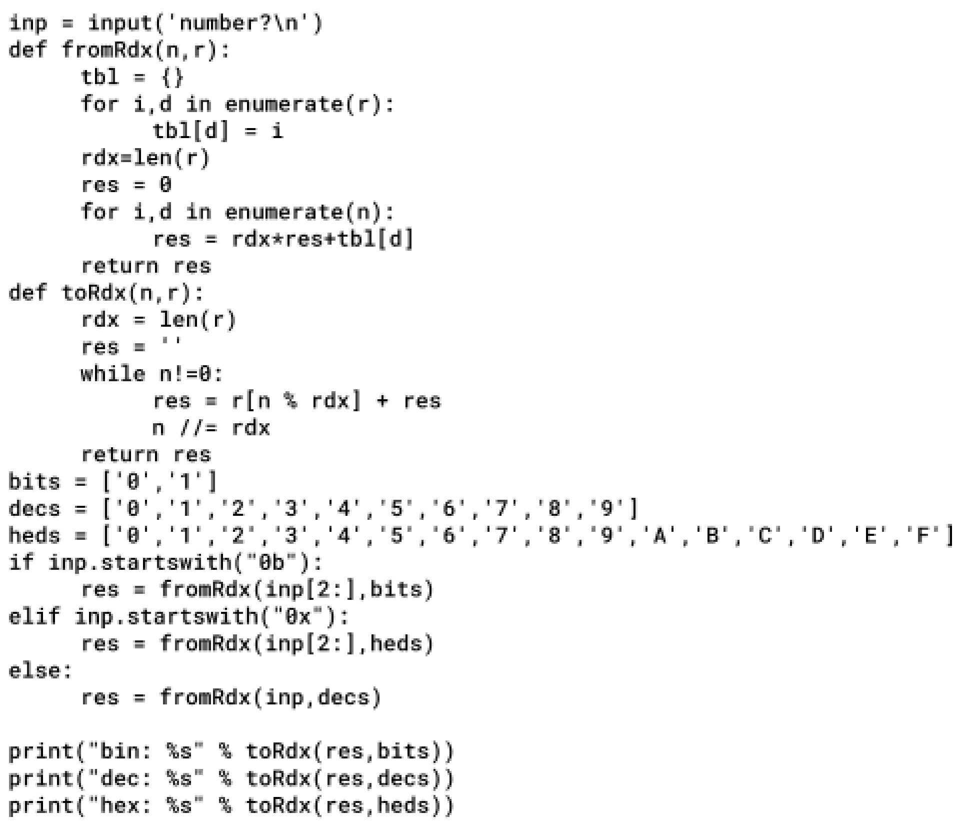
Now function fromRdx. First, create a dict with the value of each digit as a number:

```
      tbl = {}
      for i,d in enumerate(r):
            tbl[d] = i
```

Now determine the radix, which is the number of digits, and use the hint from the assignment to compute the value:

```
      rdx=len(r)
      res = 0
      for i,d in enumerate(n):
            res = rdx*res+tbl[d]
```

Every cycle, the value of the current digit is added, and the computed value of the previous digits is multiplied by the radix because they occured one place to the left. For instance, during binary conversion of 101 'res' will take on the values 1, 2 and 5. This is not a trivial snippet, I agree.



```
inp = input('number?\n')
def fromRdx(n,r):
      tbl = {}
      for i,d in enumerate(r):
            tbl[d] = i
      rdx=len(r)
      res = 0
      for i,d in enumerate(n):
            res = rdx*res+tbl[d]
      return res
def toRdx(n,r):
      rdx = len(r)
      res = ''
      while n!=0:
            res = r[n % rdx] + res
            n //= rdx
      return res
bits = ['0','1']
decs = ['0','1','2','3','4','5','6','7','8','9']
heds = ['0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F']
if inp.startswith("0b"):
      res = fromRdx(inp[2:],bits)
elif inp.startswith("0x"):
      res = fromRdx(inp[2:],heds)
else:
      res = fromRdx(inp,decs)

print("bin: %s" % toRdx(res,bits))
print("dec: %s" % toRdx(res,decs))
print("hex: %s" % toRdx(res,heds))
```

The converse is simpler. Given a number n, convert it using the given digits.

The method we learned is, to divide it by the radix and note the remainder:

```
      rdx = len(r)
      res = ''
      while n!=0:
            res = r[n % rdx] + res
            n //= rdx
      return res
```

That's mostly it. Note that prepending each digit reverses the order as is needed.

## 4.4

In 4.4 the explanation is also much tougher than the solution.
'Insert' is actually given, so your job is to understand the code.

‘Init’ just calls insert for each word in the list

```
def init(words):
     global root
     for w in words:
          root = insert(w,root)
```

Search is as follows: read the comments!

```
def search(str,tree=None):
     if tree == None: #default, search in entire tree if not given
          tree=root
     if tree == []: #reached a leaf, keyword isn’t there
          return False
     left,word,right = tree #decompose three fields
     if str==word:
          return True
     if str<word:
          return search(str,left)
     return search(str,right)
```