CS 115 Lab 6: Number Base Conversion

We have now seen multiple different number bases in class. We have seen how to convert to and from such bases, as well as how to do arithmetic between them.

However, as computer scientists, when possible, we don't want to do this work by hand. Ideally, we'd just have the computer do the conversion for us.

And that is exactly what you will do in this lab. While Python has built-ins for doing this conversion (mentioned below), knowing how to do the conversions is still useful to know. As such, you'll be implementing methods to convert numbers to their digits in another base, and back.

As a standard reminder, you are still not allowed to use loops for this lab, although we are getting very close to permitting them.

Interpreting Other Bases

Suppose I give you a number like 2025. In order to interpret this in our standard decimal system, we assign to each position in the number a value, equal to a power of 10. The number is then the sum of the digits multiplied by the appropriate power.

$$2025 = 5 \cdot 10^0 + 2 \cdot 10^1 + 0 \cdot 10^2 + 2 \cdot 10^3$$

Any number we get can be interpreted this same way, as a sum of products of digits and powers of 10.

$$4761 = 1 \cdot 10^0 + 6 \cdot 10^1 + 7 \cdot 10^2 + 4 \cdot 10^3$$

This is the direct meaning of writing a number out using its digits. And it turns out that every number[1] has a unique representation this way:

Theorem 1 (Unique Representation of Natural Numbers)**.** *Any natural number $n > 0$ can be uniquely written in the form*

$$n = a_0 \cdot 10^0 + a_1 \cdot 10^1 + \cdots + a_k \cdot 10^k,$$

where:

- *k is a non-negative integer,*
- *a_i is a digit satisfying $0 \leq a_i < 10$ for each i,*
- *and $a_k \neq 0$ (to avoid leading zeroes).*

This is the usual standard representation we use. However, it turns out that there is nothing special about the number 10 in the above theorem.

Theorem 2 (Unique Representation of Natural Numbers in Base b)**.** *For any base $b \geq 2$, any natural number $n > 0$ can be uniquely written in the form*

$$n = a_0 \cdot b^0 + a_1 \cdot b^1 + \cdots + a_k \cdot b^k,$$

where:

- *k is a non-negative integer,*
- *a_i is a digit satisfying $0 \leq a_i < b$ for each i,*
- *and $a_k \neq 0$ (to avoid leading zeroes).*

[1] positive integers only, technically, but we will only work with those for today's lab.

For example, we can observe that $101 = 81 + 9 + 9 + 1 + 1$, so we can write that

$$101 = 2 \cdot 3^0 + 0 \cdot 3^1 + 2 \cdot 3^2 + 0 \cdot 3^3 + 1 \cdot 3^4.$$

To avoid writing the above in this long form, however, we introduce shorthand notation for other bases. We use subscripts to denote that we are working in a specific base.

$$101 = 2 \cdot 3^0 + 0 \cdot 3^1 + 2 \cdot 3^2 + 0 \cdot 3^3 + 1 \cdot 3^4 = 10202_3$$

As such, given a number written in another base (for example 534_6), to find its meaning and convert it back to decimal we just have to expand the above notation:

$$534_6 = 4 \cdot 6^0 + 3 \cdot 6^1 + 5 \cdot 6^2 = 4 + 18 + 180 = 202.$$

This gives us a simple way of converting from base b to decimal.

Converting to Other Bases

But what about converting to another base? We have established earlier that any number has a unique representation in any base, but how can we find that representation?

In class, we saw a very mathematically rigorous mechanism for finding such a representation. This approach began by finding k and a_k, thus finding the most significant digit, and then finding the rest by repeating this process. This is the most convenient mechanism for converting by hand... but that is only because humans can't find remainders efficiently.

Consider for a second, the base b expansion of a number n:

$$n = a_k a_{k-1} \dots a_2 a_1 a_{0_b} = a_k \cdot b^k + a_{k-1} \cdot b^{k-1} + \dots + a_2 \cdot b^2 + a_1 \cdot b^1 + a_0 \cdot b^0.$$

Observe that everything up to the last term is divisible by b. So we can rewrite the above slightly:

$$n = a_k a_{k-1} \dots a_2 a_1 a_{0_b} = \left(a_k \cdot b^{k-1} + a_{k-1} \cdot b^{k-2} + \dots + a_2 \cdot b^1 + a_1 \cdot b^0\right) \cdot b + a_0.$$

Furthermore, recall that $0 \leq a_0 < b$. So from this, what happens when we divide n by b? In particular, what happens to its remainder? Well, as the first part is divisible by b, that actually means that the remainder is exactly a_0! So to find a_0 we just have to take n modulo b!

But that's not all. Because what now happens if we divide by b? The first part is conveniently divisible by b, but the last term is not. It is, however, smaller than b. Thus, if we then round down (say, by doing integer division), we exactly remove the last digit:

$$\begin{aligned} n//b &= \left\lfloor \frac{\left(a_k \cdot b^{k-1} + a_{k-1} \cdot b^{k-2} + \dots + a_2 \cdot b^1 + a_1 \cdot b^0\right) \cdot b + a_0}{b} \right\rfloor \\ &= a_k \cdot b^{k-1} + a_{k-1} \cdot b^{k-2} + \dots + a_2 \cdot b^1 + a_1 \cdot b^0 \\ &= a_k a_{k-1} \dots a_2 a_{1_b}. \end{aligned}$$

Then, we can proceed to use this to find the next digit as well, by repeating this same process.

To summarize, we have the following two important rules:

$$\begin{aligned} a_k a_{k-1} \dots a_2 a_1 a_{0_b} \ \% \ b &= a_0 \\ a_k a_{k-1} \dots a_2 a_1 a_{0_b} \ // \ b &= a_k a_{k-1} \dots a_2 a_{1_b} \end{aligned}$$

We can use this to implement conversion between bases, taking advantage of this recursive structure.

Task 1: Doing It Manually

Before we implement the conversions, we should try to understand this process by doing it ourselves.

To begin with, try to convert the integer 1234 (as always, if we omit an explicit base, treat the number as being in base 10) into base 5. Make sure you note down your steps explicitly.

Note down your work in your `lab6_report.txt`. In particular, we will be grading for the presence of a process, and based on the approach you took. We will not be grading you on the correctness of your answer, or on using a specific method. This should be taken as an opportunity to practice, and we want to see you attempting to use the methods, and to explain your understanding of them.

CA Check-In

After you complete the above task, talk to the CA in your section. Show them your calculations, and explain the process you took.

Task 2: Conversion from Other Bases

Now, it is time to code. We begin by asking you to implement the `to_int` function. This takes in a number in some base and a base, and returns the integers this corresponds to.

In order to do this, we must have some way of representing a number in another base. As a number is just a sequence of digits, we thus will represent it as a sequence of digits: A tuple! For example, we can represent 1321_4 as the tuple `(1, 3, 2, 1)`, listing the digits of the number in order. Note that we don't include the base, and that information must be stored elsewhere.

Your `to_int` function must take in a `tuple` of digits and a base, and return the number these digits represent in that base. You may assume that the tuple will be non-empty, and that the digits will always be numbers in the range `0` to `base-1`.

Note that the digits will be given with the most significant digit first! In other words, `digits[0]` is a_k, not a_0! Keep that in mind while writing your code.

Some sample input-output pairs are listed below.

```
>>> to_int((1, 1, 1), 4)
21
>>> to_int((1, 2, 12), 16)
300
>>> to_int((1, 0, 2, 0, 2), 3)
101
```

Task 3: Conversion to Other Bases

After this, you now want to go backwards, implementing the more challenging task. We want you to implement the `to_base` function, which does the reverse of `to_int`.

`to_base` takes in a positive integer `n` and a base, and returns a `tuple` with the digits of its representation in the given base. As before, these digits need to be in the valid range, and the most significant digit has to be first.

Note that this function is only defined for positive integers. We don't specify what to do when `n` is 0. This is something that you may decide for yourself, and we will not be testing it. Your function may return whatever you feel is reasonable.

Some sample input-output pairs are listed below.

```
>>> to_base(21, 4)
(1, 1, 1)
>>> to_base(300, 16)
(1, 2, 12)
>>> to_base(101, 3)
(1, 0, 2, 0, 2)
```

Task 4: Analysing Your Functions

While we have now written our own conversion functions, Python actually has its own built-ins. Note that you are not allowed to use these to implement your functions, but we do want you to play around with them to better understand them.

First, recall that `int` can be used to convert strings into integers. For example, `int("7037")` returns `7037`.

But the `int` method actually can take an optional second argument, which represents a base. This lets you parse strings representing a number in some other base. For example, `int("10202", 3)` will interpret and decode the number 10202_3, thus returning `101`.

Going the other way is a bit harder. There is no universal built-in function for converting numbers to arbitrary bases. However, there are a few useful built-ins for specific bases:

- `bin`: Converts a number into a binary string. For example, `bin(17)` returns `"0b10001"` (the `0b` part being an indicator that this is a binary number).
- `oct`: Converts a number into an octal string. For example, `oct(75)` returns `"0o113"`.
- `hex`: Converts a number into a hexadecimal string. For example, `hex(300)` returns `"0x12c"`. Note that this uses the characters `a` through `f` for the larger digits.

These functions can be used to verify your `to_int` and `to_base` functions. You can, and should use these by comparing the output of your functions with the correct answer given by the built-ins.

In particular, we require you to convert 48879 into base 16. You should do this using both your own `to_base`, and using the built-in `hex` function. Note the difference in the output style, and note any differences in your lab report.

Once you have convinced yourself that your code is correct, you should also recheck your calculations for task 1. Note in your report whether your calculations were correct or not.

Submission Instructions

For this lab, you must submit the following files, with exactly the following names:

- `lab6.py`
- `lab6_report.txt`

Remember to put your name and the Stevens Honor Pledge on every file you submit! Failure to put the pledge can result in marks being deducted, with repeat penalties potentially getting stricter between assignments.

Submissions must be handled through Gradescope (accessible through the homework page on Canvas). Grades will be released through Gradescope once the homework is graded.

Rubric

Category	Description	Points
Name	For stating your name on every file for submission.	1
Pledge	For writing the full Stevens Honor Pledge on every file for submission.	2
Collaborators	For filling out the Collaboration Statement.	2
Check-In	For performing the CA Check-In during the lab.	20
Task 1: Manual Calculations	For having detailed base conversion steps in your report.	10
Task 2: `to_int`	For having a correctly working `to_int` function.	30
Task 3: `to_base`	For having a correctly working `to_base` function.	30
Task 4: Analysis	For filling out the analysis questions in the report.	5
Total		100

Notes and Reminders

- **Remember to include docstrings!** If your submission does not have docstrings, or has docstrings that are not related to your code, you will not get credit for them.
- We have provided a test file for you to test your solutions. By running this test file, you should be able to verify that your files are named properly, and that your code works correctly.
- Test frequently! Try out many different cases to verify that your code runs. Our provided test file may not be comprehensive, and so you should ensure to test your code on many other cases as well.
- Code that doesn't compile (throws a `SyntaxError` on being run) will receive a 0 for the assignment! Make sure you submit code that can be run, even if it does not pass all the test cases.
- **Don't forget to add your name and pledge!**