CSCI 1300/5001 – Fall 2019 Handout: Error Checking
Introduction to Object-Oriented Programming
Michael H. Goldwasser
Saint Louis University Monday, 14 October 2019

Exploration of Number Systems

Part 1: Number Systems

While our current programming lesson involves error checking, we begin this exploration by sneaking in a valuable math lesson for computer scientists, namely about number systems used for representing integers.

Humans are used to the **decimal ("base ten") number system** in which each character digit represents a multiple of some power of ten (presumably because we have ten fingers). For example, we use the representation `2019` for the quantity we might otherwise describe as *two-thousand and nineteen*. More formally, in the decimal system, the rightmost digit is the "ones digit" and so the numeral "9" in that place represents the quantity *nine*. To its left is the "tens digit" and the numeral given there represents an appropriate multiple of ten (in this case $1 \times 10 = 10$). To the left of that is the hundreds digit (as $10^2 = 100$), although in our example above, the digit "0" represents a contribution of zero hundreds, and the leftmost digit of this example is a "2" that represents $2 \times 10^3 = 2000$. Returning to the context of Python and programmer, we have already seen that the **int** class constructor can assist us in converting strings of characters to the underlying integer values, for example with **int**`('2019')` returning the actual value we know as `2019`.

In computer science, there are a lot of two-state systems (e.g. true/false, on/off) and so there is a great interest in the **binary ("base two") system** in which the places are powers of two. The rightmost digit again represents ones, but the digit immediate to its left represents twos, the next represents fours, then eights, then sixteens, and so forth. In a binary system, each individual digit is either a "0" or a "1"; as with the decimal system, if we need to increase a digit beyond that we can instead "carry" to the next higher power of two. So in a binary system, the string "10011" represents the quantity known as *nineteen* in our decimal system as it could be viewed as $1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 16 + 2 + 1 = 19$.

1. What is the decimal value of the binary number `110100`?

2. How would the decimal value `41` be represented in binary?

Computer scientists also tend to be interested in other bases that are powers of two, most typically octal ("base eight") and hexadecimal ("base sixteen"), for reasons we will explain in a moment. Starting with octal, we note that the place values are powers of eight with the rightmost being $8^0 = 1$, the second from the right being $8^1 = 8$, the next being $8^2 = 64$. Then each digit can be any value from 0 through 7. As an example, the octal value `314` would actually be equal the the value familiar to us as 204 in decimal (i.e., two-hundred and four). To see this, we note that $3 \cdot 8^2 + 1 \cdot 8^1 + 4 \cdot 8^0 = 3 \cdot 64 + 1 \cdot 8 + 4 \cdot 1 = 192 + 8 + 4 = 204$.

3. What is the decimal value of the octal number `246`?

4. How would the decimal value `53` be represented in octal?

Why computer scientists have an affinity to a system like octal is because eight is a power of two and that makes it very convenient to convert between binary and octal (in fact, far easier that to convert either of those to or from decimal). For larger numbers, it is quite difficult for us humans to effectively remember, transcribe, or communicate long binary values such as `1011101101011`; there are simply a lot of ones and zeros. Try memorizing that, or even reading it to a friend and letting them write it down. However, a binary value can easily be converted to octal by considering groups of three binary digits (bits) starting at the rightmost. Every three bits corresponds to a single octal digit (as three bits can represent any number from 0 to 7). So the value `1101001100011` can be though of in three-bit chunks as `1 101 001 100 011` (note well we started grouping at the right side not the left). In octal, that same number is represented as `15143`, and perhaps that's a bit easier for us to memorize or transcribe.

5. Convert the following binary value to octal: `1111101010110100`

6. Convert the following octal value to binary: `4725`

Part 2: Python conversions

We noted earlier that Python's **int**( ) constructor will compute the numeric representation for a given string of (character) digits. By default, it does assume a decimal representation, and thus **int**('314') does produce the integer we describe as three-hundred and fourteen. However, that constructor accepts a second (optional) parameter which is an integer that defines the base of the number system. For example, the expression **int**('314',8) produces the octal interpretation of that value, which as we discussed earlier is the decimal 204. Similarly, the Python expression **int**('10011',2) produces the integer value nineteen.

Of course, the integer constructor also has to cope with what happens if invalid parameters are sent to it.

7. What happens if calling **int**('FAB4')? Why?

8. What happens if calling **int**('391', 2)? Why?

9. What happens if calling **int**('391', 8)? Why?

Part 3: Our implementation

While this functionality of converting numbers is already supported by Python's **int** constructor, let's develop our own implementation. To avoid confusion, we'll name ours toDecimal an assume a signature such as toDecimal(s, base) where s should be a character string representing the digits and base should be an integer that for the sake of this context should be at least 2 and at most 10.

We want your focus to be on the interesting combination of expectations and how to validate the parameters and raise appropriate expectations when the expectations are not met. The actual math of the conversion is also interesting, but we'll provide you with a working implementation (that is, so long as the user sends valid parameters).

```
def toDecimal(s, base):
    total = 0
    for k,digit in enumerate(s):
        total += int(digit) * base**(len(s)-1-k)
    return total
```

Experiment with our implementation of the function, paying particular attention to what happens when unexpected parameters are sent.

10. What happens if calling `toDecimal(314, 8)`? Explain your observations.

11. What happens if calling `toDecimal('314', '8')`? Explain your observations.

12. What happens if calling `toDecimal(8, '314')`? Explain your observations.

13. What happens if calling `toDecimal('314', 2)`? Explain your observations.

14. What happens if calling `toDecimal('314', −2)`? Explain your observations.

15. What happens if calling `toDecimal('314', 16)`? Explain your observations.

16. What happens if calling `toDecimal('FAB4', 16)`? Explain your observations.

17. After reflecting on the above scenarios, give an enumerated list of specific expectations about parameters `s` and `base` and their relation to each other. (Feel free to add more items to the list as you see fit.)

    -
    -
    -
    -
    -

18. When validating parameters, the order in which you validate your expectations is quite important. What is wrong with an implementation that begins as follows?

```
def toDecimal(s, base):
    if base < 2 or base > 10:
        raise ValueError('invalid base')
    ...
```

19. Return to the list of conditions you developed on the previous page and list those again, this time in an appropriate order in which the validations should be performed.

    -
    -
    -
    -
    -
    -
    -

20. Revise our original implementation to perform robust validation of all parameters. Test your new implementation on all the test cases given on page 4.

Part 4: Hexadecimal (Bonus time)

We originally mentioned that computer scientists sometimes work with bases that are powers of two such as octal (base 8) and hexadecimal (base 16), because conversion between binary and these powers is trivially done by converting groups of bits. Computer systems often have hardware that physically process bits in groups of 16 bits, 32 bits, 64-bits, or even 128 or 256 bits at a time. Although octal is convenient, because an octal digit corresponds to three binary digits, a 16-bit binary value needs more than five octal digits but doesn't really need a full six octal digits.

Hexadecimal (base 16) uses similar conventions as other bases, with the rightmost digit representing a multiple of $16^0 = 1$, the second rightmost digit representing a multiple of $16^1 = 16$, the next representing $16^2 = 256$ and so forth. However, for an individual digit in hexadecimal, we need to be able to represent any quantity from 0 to 15, since we cannot "carry" to the next higher power until reaching 16. When writing a hexadecimal string, we wish to use a single character for each digit. Clearly, we can use the characters "0" through "9" for the conventional values zero through nine. To extend beyond those, computer scientists started using characters of the alphabet such that "A" represents digit value ten, "B" as eleven, continuing to "F" as fifteen.

As an example, the hexadecimal string `'FAB4'` represents the decimal value 64180 as it is $15 \times 16^3 + 10 \times 16^2 + 11 \times 16^1 + 4 \times 16^0 = 15 \times 4096 + 10 \times 256 + 11 \times 16 + 4 \times 1 = 15 \times 4096 + 10 \times 256 + 11 \times 16 + 4 \times 1 = 61440 + 2560 + 176 + 4 = 64180$.

You could also have Python do the conversion for you as **int**(`'FAB4'`, 16). Python also accepts the corresponding lowercased characters in hexadecimal, thus **int**(`'fab4'`, 16).

21. Extend the implementation of the toDecimal function to allow for the use of bases such as 16. In fact, you should allow bases up to 36, with letters A through Z representing digits 10 through 35, while rigorously validating that the string literals given are appropriate for the indicated base.

    Hint: For character c from `'0'` to `'9'`, we relied on syntax **int**(c) to get the integer represented by that symbol. For uppercase letters `'A'` through `'Z'`, the corresponding digit value can be computed using the expression 10 + ord(c) − ord(`'A'`).

    (We could explain why, but just trust us for now.)