Lecture 04

Program Input and Number Systems

2 Jour du Génie, Year CCXXX

Song of the day: ***Rally, Rally (fear. Pii & meiyo)*** *by MAISONdes (2021).*

Sections:

Part 0: *Review*

Let's start with a quick review problem. Let's pretend we have two classroom sizes: one that fits 35 students and one that fits 15. Write a program that does the following:

1. Ask the user how large the student body is (i.e. how many students there are).
2. Determine how many 35-student classrooms we can form with this many students.
3. Determine how many 15-student classrooms we can form with the remaining students.
4. Display the results of steps 2 and 3, along with how many students remain leftover.

Number 1 is an easy one; we use the `input()` and `int()` functions. I'm also going to define two variables to store the sizes of our classrooms, so that I can keep track of them and change them at any point if I so wish:

```
class_size_a = 35
class_size_b = 15

num_of_students = int(input("How large is the student body? "))
```

Now, for step 2, I'm going to use the same technique we used when we wanted to see how many quarters we could form with a specific amount of pennies. This time, though, it's not pennies but students, and it's not 25-cent groups, but 35-student groups. For this, we use the `//` operator:

```
num_size_a = num_of_students // class_size_a
```

How can we determine how many students remain after this operation? The `%` operator, which gives us the remainder after a division, should do the trick:

```
num_of_students = num_of_students % class_size_a
```

Using this amount of remaining students, we can see how many 15-student classrooms we can form by literally repeating the same process using `class_size_b` instead of `class_size_a`:

```
num_size_b = num_of_students // class_size_b
num_of_students = num_of_students % class_size_b  # this is the number of
leftover students
```

Finally, step 3 just requires a quick `print()` statement:

```
print("We formed " + str(num_size_a) + " 35-student classroom(s), " +
str(num_size_b) +
      " 15-student classrooms, and have " + str(num_of_students) + "
leftover students.")
```

Here's the full solution.

Part 1: *Number Systems*

What do we mean when we say "magnitudes bigger than", or "magnitudes smaller than"? Mathematically speaking, for something to be a magnitude larger or smaller than another thing, they have to follow the same number system (i.e. you can't compare apples to oranges, etc.)

Since the world today primarily uses the **decimal system** for mathematical calculations, when we say a is (for example) "3 magnitudes" larger" than b, this is what we mean:

> $a = b * 10^3$

So, if b equal **42**, a would equal **42,000**.

In other words, the number we use to differentiate magnitudes in the decimal, or **base-10**, system is the number **10**. We can also tell that this is the case because **there is no single digit to represent 10**. We, instead, have to write out ten as a combination of two digits, 1 and 0.

The same is the case for 100. Since we don't have a symbol for 10, we cannot represent 10 tens in a two digit format. Therefore, we need to use three digits to go up a magnitude (1, 0, and 0).

Another way of thinking of base-10 numbers is as the **sum of numbers multiplied by the powers of 10**:

> $4{,}034 = 4 * \mathbf{10^3} + 0 * \mathbf{10^2} + 3 * \mathbf{10^1} + 4 * \mathbf{10^0} = 4000 + 0 + 30 + 4$

But what about other number systems? Computers, for instance, do use the decimal system to count or do mathematical operations. The reason for this is that computers can only do operations in **ones and zeros**. This number system uses **2** instead of 10 to differentiate between magnitudes. We would, thus, call this system ***binary***, or ***base-2***.

To count from 1 to 10 in binary, then, we would do the following:

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

***Figure 1**: Counting to 10 in **binary**.*

Just like decimal assumes that we don't have a symbol for 10, **binary assumes that we don't have a symbol for 2**. Therefore, we must count using only the numbers under **2** (i.e. 0 and 1).

Again, just like decimal can be represented as a sum of numbers being multiplied by the powers of 10, **binary numbers can be represented as a sum of number being multiplied by the powers of 2**. For example:

> $(1001)_2 = 1 * \mathbf{2^3} + 0 * \mathbf{2^2} + 0 * \mathbf{2^1} + 1 * \mathbf{2^0} = 8 + 0 + 0 + 1 = (9)_{10}$

We can better illustrate this with the following **web tool**:

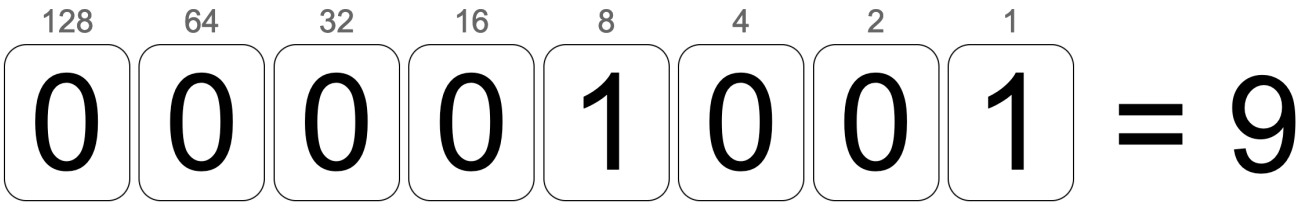


***Figure 2**: A visual representation of $(1001)_2$ to $(9)_{10}$.*

If you're given a number in decimal, and have to convert to binary, you can do a simple division and take note of the remainders, which end up being your binary number. For example, if we wanted to convert 29 to binary:

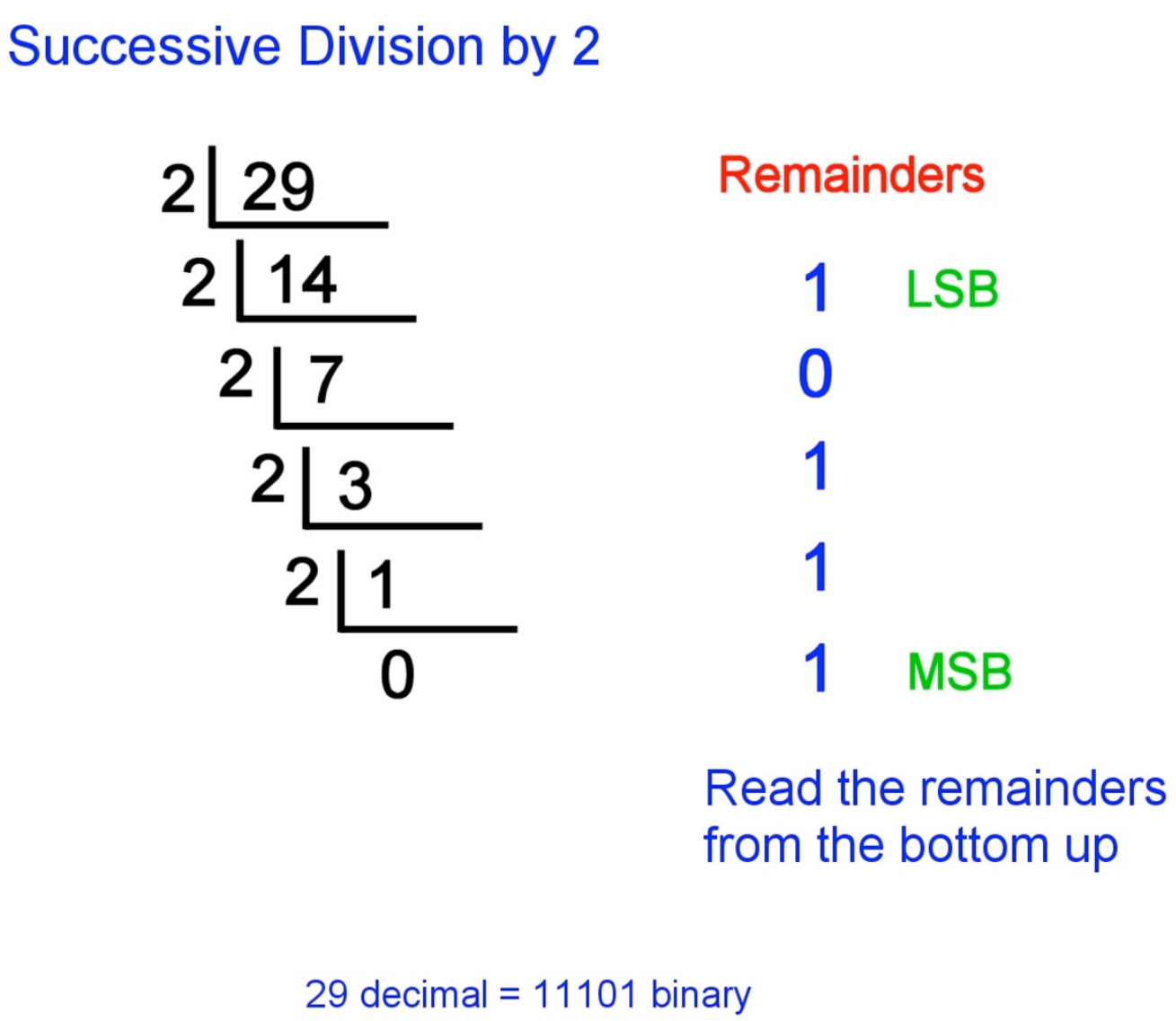


***Figure 3**: Converting $(29)_{10}$ to $(11101)_2$ (**source**).*

You can do this process for just about any number system. Let's try a base-5 system for funsies.

Decimal	Base-5
0	0
1	1
2	2
3	3
4	4
5	10
6	11

Decimal	Base-5
7	12
8	13
9	14
10	20
11	21
12	22
13	23
14	24
15	30
16	31
17	32
18	33
19	34
20	40
21	41
22	42
23	43
24	44
25	100
26	101
27	102
28	103
29	104
30	110

***Figure 4**: Counting to 30 in base-5.*

Not all number systems are equally relevant to computer science, however. The second most important base to be aware of is base-16, or **hexadecimal**.

Decimal	Hexadecimal
0	0

Decimal	Hexadecimal
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	a
11	b
12	c
13	d
14	e
15	f
16	10
17	11
18	12
19	13
20	14

***Figure 5**: Counting to 20 in **hexadecimal**.*

As you can see, because in Arabic numerals we don't have a symbol for anything larger than 9, and because we're not allowed to use 10 until go up in magnitude, we have to resort to using tokens for these values. You can literally use whatever symbol you want as a token, but the well-established convention is to use the letters `a` through `f`.

Why is hex relevant to computer science?

Memory is stored inside your machine in units called **bytes**, which are themselves usually composed 8 bits of **bits** (a 1 or a 0). Let's say we have one such byte below:

```
10111001
```

The reason why hexadecimal is helpful here is that, if we split this byte into two units of 4 bits:

```
1011 1001
```

Each of those 2 halves can give us a **total of 15 possible bit combinations** before growing in size (magnitude):

1. `0000`
2. `0001`
3. `0010`
4. `0011`
5. `0100`
6. `0101`
7. `0111`
8. `1000`
9. `1001`
10. `1010`
11. `1011`
12. `1100`
13. `1101`
14. `1110`
15. `1111`

What system can only count to 15 before growing in magnitude? Hexadecimal! This, consequently, makes conversions between decimal and hexadecimal extremely simple *if you know your binary equivalents*:

Let's say we wanted to convert $(1967)_{10}$ to both binary and hex.

Using repeated division by 2, we get:

> 1967 / 2 = 983.5 —> **1**
>
> 983 / 2 = 491.5 —> **1**
>
> 491 / 2 = 245.5 —> **1**
>
> 245 / 2 = 122.5 —> **1**
>
> 122 / 2 = 61 —> **0**
>
> 61 / 2 = 30.5 —> **1**
>
> 30 / 2 = 15 —> **0**
>
> 15 / 2 = 7.5 —> **1**
>
> 3 / 2 = 1.5 —> **1**
>
> 1 / 2 = 0.5 —> **1**

Reading from the bottom up, we get **$(11110101111)_2$**. Now, split this binary number into groups of four

> **0111 1010 1111**

The decimal equivalents of these three groupings are easy to figure out: **7**, **10**, and **15**. By the same token, finding the hexadecimal equivalents of 7, 10 and 15 isn't difficult if we are familiar with counting in hex (figure 4):

> $(1967)_{10} = (11110101111)_2 = \mathbf{(7af)_{16}}$.

And just like that, we converted 1967 to all relevant units in one fell swoop!

Part 2: *The `math` Module*

You know how, in a previous lecture, I asked you to calculate the volume of a cone? For many mathematical operations, we need to use certain, pre-defined constants, such as **pi**. In our case, I asked you to define a variable that would hold your best estimation of this value:

```
pi = 3.14156  # for example
```

It might not come as too much of a surprise that approximating such common and important constants is very bad practice. This is especially the case because programming is often used in engineering applications where precision is of paramount importance. In other words, you are not going to tell your boss and NASA that you programmed a rover by "sort of guessing the value of pi." The great thing is that you really don't have to at all!

One of the great things about Python is that it has a ***huge*** community that constantly releases their code to the public—free of charge—for us to use. When we want to make use of this code, we have to import it in the form of a ***module***.

One of the most common modules is the `math` module which, as you can probably guess, contains a plethora of math related functions and values that we can use:

```
import math

pi = math.pi
e = math.e

print(pi)
print(e)
print(math.sin(pi))  # prints the sine of pi
print(math.sqrt(e))  # prints the square-root of e
print((math.pow(pi, e)))  # prints pi ** e
print(math.radians(pi))  # prints the radian equivalent of pi degrees
print(math.floor(e))  # rounds e up
print(math.ceil(e))  # rounds e down
```

Output:

```
3.141592653589793
2.718281828459045
1.2246467991473532e-16
1.6487212707001282
22.45915771836104
0.05483113556160755
2
3
```

As you can see, we need to explicitly import the module for Python to be able to use it (`import math`). Note the format of module function calls:

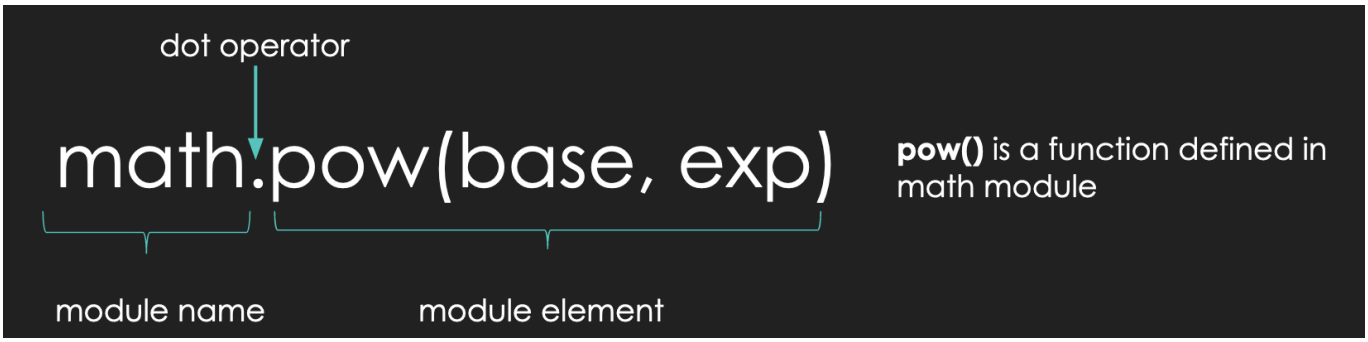


Figure 1: The format of a function call from the `math` module.

So, if we were to calculate the volume of our cone again—properly this time—I would now do something like **this**:

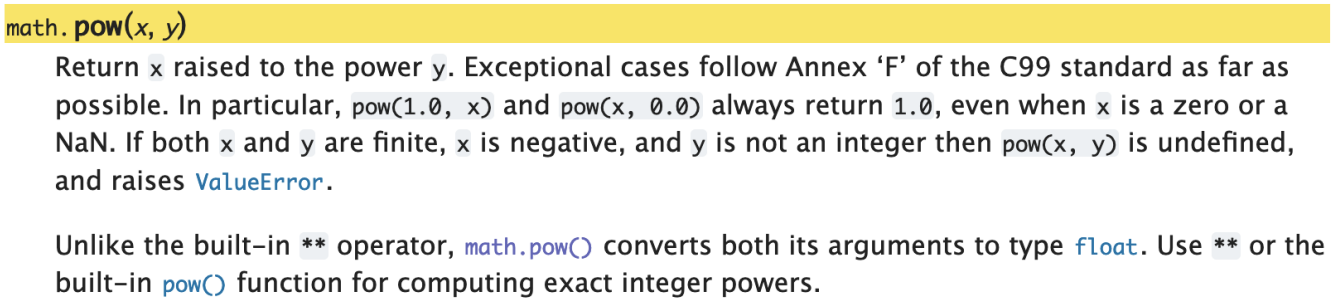
```
import math

base_radius = float(input("Please enter the length of the cone base
radius: "))
cone_height = float(input("Please enter the length of the cone height: "))

constants = math.pi / 3
variables = math.pow(base_radius, 2) * cone_height  # the use of
math.pow() is not strictly necessary, but I'm proving a point

volume = constants * variables

print("The volume of this cone is " + str(volume) + ".")
```

Code Block 1: A better **solution** for our cone volume problem.

Notice here that, when I used `math.pi`, I did not follow it with a set of parentheses `()`. This is because **`pi` is not a function** (like `print()`, `input()`, etc.), but rather a simple value. On the other hand, we can see that the `math.pow()` function call makes use of parentheses. This is because all Python function calls

require the use of parentheses. We will learn more about the specific of functions after the first midterm, but for now, you can safely assume that this is always the case.

According to the `math` module documentation, inside `math.pow()`'s parentheses, you must put the value of the base that you want to raise, and the power to which you want to raise it, in that order:



`math.` **pow**(*x*, *y*)

Return `x` raised to the power `y`. Exceptional cases follow Annex 'F' of the C99 standard as far as possible. In particular, `pow(1.0, x)` and `pow(x, 0.0)` always return `1.0`, even when `x` is a zero or a NaN. If both `x` and `y` are finite, `x` is negative, and `y` is not an integer then `pow(x, y)` is undefined, and raises `ValueError`.

Unlike the built-in `**` operator, `math.pow()` converts both its arguments to type `float`. Use `**` or the built-in `pow()` function for computing exact integer powers.

Figure 2: `math.pow()`'s documentation, explaining its use and its difference from the built-in `**` operator.

Here's the entire **documentation** for the `math` module for your reference.

Part 3: *The `random` Module*

Another very common module is the `random` module. It basically is what it sounds: a library of functions that deal with (pseudo-)random behavior.

The most basic of these is the `random()` function, which always returns a pseudo-randomly generated decimal `float` value:

```
import random

random_decimal = random.random()
print(random_decimal)

random_decimal = random.random()
print(random_decimal)

random_decimal = random.random()
print(random_decimal)

random_decimal = random.random()
print(random_decimal)
```

A possible output:

```
0.6549562234417277
0.8773055016457298
0.6249540159645146
0.5591596841328375
```

How would this be useful? The most basic example I can think of is a **coin-flip program**, where `1` is heads and `0` is tails:

```
import random

random_decimal = random.random()
result = round(random_decimal)

print("The result of this coin flip is: " + str(result))
```

Code Block 2: **Coin flipping** with the `random` module.

A possible output—it has roughly a 50-50 chance of being either a `1` or a `0`:

```
1
```

Note: The `round()` function simply rounds a number to its closest integer value.

If you would like to instead generate random integers, we could make use of the `randrange()` function:

```
import random

lowest_possible = 1
upper_limit = 10

random_integer = random.randrange(lowest_possible, upper_limit)

print(random_integer)
```

Code Block 3: **Generating** random integers.

A possible output:

```
8
```

The `randrange()` function takes two arguments (i.e. values inside the parentheses). The first value represents the lowest possible integer that can be returned. The second value marks the upper limit—this means all possible numbers ***below*** this value are possible. In other words, the upper limit is **non-inclusive**. This being the case, our code above can produce any integer value between 1 and 9.

If this "limitation" sounds weird to you, don't worry—it *is* weird. In fact, there's actually another function in the `random` module, `randint()`, where both values are inclusive. The reasons for `randrange()` will become obvious a bit later in the semester, but when it comes to this module, feel free to use either or both unless instructed otherwise:

```
import random

lower_limit = 1
upper_limit = 10

random_integer_a = random.randrange(lower_limit, upper_limit)
random_integer_b = random.randint(lower_limit, upper_limit)

print("A random number from", lower_limit, "(inclusive) and", upper_limit,
"(exclusive):", random_integer_a)
print("A random number from", lower_limit, "(inclusive) and", upper_limit,
"(inclusive):", random_integer_b)
```

Possible output:

```
A random number from 1 (inclusive) and 10 (exclusive): 3
A random number from 1 (inclusive) and 10 (inclusive): 10
```

For now, these are the functions from the `random()` module that you will be using the most, but we will be getting into others later in the semester.

Here's the `random` module's **documentation** for your reference.