# Lecture-Notes-Sep-8-binary-representations

September 12, 2022

## 1 Lecture 2

- A breif review of Python variables
- Float point real numbers and round-off errors

## 2 Variable

A placeholder (a block of computer memory) that stores a value. It is called a variable because its value may change (through value assignment).

### 2.1 Name

Variables are referred to by their names. The names are letters (a-z and A-Z) followed by other letters or digits or the underscore _. They are case sensitive, i.e., a and A are two different variables.

### 2.2 Value

A variable holds a single value. They can be one of the following types * **a number** * **an interger**, like 1, 2, 1000 etc. There is effectively no limit on how large the integer can be, as long as the computer has enough memory to hold it. * **a float-point number**, like 1.3, -10/3, etc, or in scientific notation $-5.3 \times 10^{-8}$ can be written as -5.3e-8 or 5.3E-8, i.e., a value followed by **e** or **E** followed by an exponent of 10. * **a complex number**, such as 1+2j (which is commonly known as $1 + 2i$) where 1 is the real part and 2 is the imaginary part. * We can use complex(1, 2) to construct it, or simply 1+2J * We can use arithmatic operations such as +, -, *, /, **.

[1]:
```
# example what is 5e6 equal to?
print("5e6=", 5e6)
# complex number example
# set the value of the value C to 1+2j
C = 1+2j
print("C=", C)
print("C == complex(1, 2)?", C == complex(1, 2))
# the real part and imaginary part of C can be accessed as C.real and C.imag
print("The real part of C =", C.real)
print("The imaginary part of C =", C.imag)
print("(1+2j)*(2+1j)/(3-1j)=", (1+2j)*(2+1j)/(3-1j))
```

```
5e6= 5000000.0
C= (1+2j)
```

```
C == complex(1, 2)? True
The real part of C = 1.0
The imaginary part of C = 2.0
(1+2j)*(2+1j)/(3-1j)= (-0.49999999999999994+1.5j)
```

The last answer is peculiar. If we calculate by hand, we can verify that the true answer is $-0.5+1.5j$. Why is it a tiny bit off? This is the rounding error, that we will cover later in this lecture

- **a string** is a string of characters, either single quoted, like 'this is a string', or double quoted "this is another string". Single or double quoted strings are the same as long as the strong of characters are the same.
  - We can compare them, using ==, >, <, >= or <=
  - We can concatenate them using +

[2]:
```
s = "it's correct"
print("s=", s)
# the operator == compares two values
print("s == 'test'?", s == 'test')
print("s + '!' =", s + '!')
```

```
s= it's correct
s == 'test'? False
s + '!' = it's correct!
```

- **a boolean**, a logical value being either True or False
  - we can compare them using ==, !=
  - we can perform boolean operations and, or, not

[3]:
```
# boolean example
a = 2 > 1
print("2 > 1 = ", a)
print("a != True: ", a != True)
x = 3
print("x>0 and x < 10 =", x>0 and x<10)
```

```
2 > 1 =  True
a != True:  False
x>0 and x < 10 = True
```

- Other types, such as functions (e.g., print as we have used), lists, dictionaries or classes. We will see them later in this course.

**Unlike typed languages**, Python variables can hold any type of values.

[4]:
```
s=12 # a number
print("s=", s)
s="a" # the same variable, now holds a string
print("s=",s)
```

```
s= 12
s= a
```

## 3 A Simple Example: Free Falling Object

For a free falling object, given an initial positive $x_0$ and an initial velocity $v_0$ at time 0, what is its position $x(t)$ at time $t$?

$$x(t) = \frac{1}{2}gt^2 + v_0 t + x_0$$

[5]:
```
# Inputs:
# the gravitational constant g
# Python functions are organized in to modules

# the constant g is defined the scipy model, constants subpackage
from scipy.constants import g

# time t
t = 2

# initial velocity
v0 = 2

# initial positive
x0 = 1

# positive
x = g/2 * t**2 + v0 * t + x0
print("t =", t, "x =", x)

# at a different time 3
t = 3
print("t =", t, "x =", x, "what?")
# Why does the value of x does not change?
# how to we compute x(3)?
```

```
t = 2 x = 24.6133
t = 3 x = 24.6133 what?
```

[6]:
```
x = g/2 * t**2 + v0 * t + x0
print("t = 3, x =", x, "correct!")
```

```
t = 3, x = 51.129925 correct!
```

## 4 Based of Numbers

### 4.1 Base 10

For example,

$$321 = 3 \times 10^2 + 2 \times 10 + 1$$

### 4.2 Other bases

We can extend this idea to any base:

A number $N$ in base $B$, with digits $0 \leq d_i < B$ for $i = 0, \ldots, m$, is

$$N = \sum_{i=0}^{m} d_i B^i$$

We typically write is as $(d_m d_{m-1} \ldots d_2 d_1)_B$

**Example** How do we express $(321)_{10}$ in base 8?

- Let us start with the lowest digit, which is the remainder of 321/8?
  - the remainer is calculated by the modulus operator `%`

[7]:
```
# 321 in base 8
N = 321
B = 8
# The last digit
d0 = N % B
print("d0 =", d0)
```

```
d0 = 1
```

- For the second last digit, please note that,

$$321 = 40 \times 8 + 1$$

  - After we divide 321 by 8, we shift the digits to the right by one position
  - Here the quotient 40 (without the remainder) is composed of the remaining digits.
  - The second digit is thus the lowest digit of the quotient
  - Thus, we can get the last digit of the remainder 40, which is the second last digit of 321

[8]:
```
# second last digit
# we get the quotient first, here we use the floor division `//`, which ignores␣
 ↪the fractional part.
q = N // B
print("q=", q)

# then d1 is the last digit of q
d1 = q % B
print("d1 =", d1)
```

```
q= 40
d1 = 0
```

- We then repeat to get the remaining digits one by one

[9]:
```
# the thrid last digit
# we get the quotient first
q = q // B
```

```
print("q=", q)
# then d2 is the last digit of q
d2 = q % B
print("d2 =", d2)

# the fourth last digit
# we get the quotient first
q = q // B
print("q=", q)
# then d3 is the last digit of q
d3 = q % B
print("d3 =", d3)
```

```
q= 5
d2 = 5
q= 0
d3 = 0
```

- When do we stop?

$$(321)_{10} = 5 \times 8^2 + 0 \times 8 + 1 = (501)_8$$

- This is a tedious process. But each step is repetitive. It is a good candidate for using loops, which we will learn next week!

# 5 Fractions

The same idea can be extended to fractions, a real number

$$R = \sum_{i=-\infty}^{m} d_i B^i$$

For example, in base 8,

$$(9.3)_{10} = 1 \times 8 + 1 + 2 \times 8^{-1} + 3 \times 8^{-2} + 1 \times 8^{-3} + 4 \times 8^{-4} + 6 \times 8^{-5} + \cdots = (11.2\overline{3146})_8$$

**A finite digit fraction in base 10 may have infinite digits in other bases**

## 5.1 How do we convert a fraction to other bases?

- We have already dealt with the integer part. Lets just consider $0 < R < 1$

$$R = \sum_{i=-\infty}^{-1} d_i B^i$$

- Lets start with a example: $(0.12345)_{10} \times 10 = (1.2345)_{10}$
  - we multiply the number by the base
  - the integer part of the product is the first digit

  - the other digits are shifted to the left
- In general, multiply $R$ by $B$, then

$$BR = d_{-1} + \sum_{i=-\infty}^{-1} d_i B^{i+1} = d_{-1} + \sum_{i=-\infty}^{-1} d_{i-1} B^i$$

  - So, the integer part of $BR$ is $d_{-1}$
  - The second digit of $R$ becomes the first digit of $BR$

[10]:
```
# How do we convert
R = 0.3
B = 8

d_1 = int(B*R) # here we convert B*R into an integer, i.e., ignore the fraction␣
 ↪part.
print("d_1 =", d_1)
R = (B*R) % 1 # modulus 1 (i.e., % 1) take the fraction part.
print("R =", R, "not equal to 4!") # here , R = 0.3999999999999999, the true␣
 ↪value should be 0.4
# this is our first taste of numerical errors. We will see why in a moment
```

```
d_1 = 2
R = 0.3999999999999999 not equal to 4!
```

- We repeat the above step to get the first digit of $BR$, which is $d_{-2}$, the second digit of $R$

[11]:
```
d_2 = int(B*R)
print("d_2 =", d_2)
R = (B*R) % 1 # modulus 1 (i.e., % 1) take the fraction part.
print("R =", R) # here , R = 0.3999999999999999, the true value should be 0.4
```

```
d_2 = 3
R = 0.1999999999999993
```

- We repeat this process until either $R = 0$ or enough digits is obtained

**This is a tedious process** This is why we need loops to automate the repetition.

[12]:
```
R = (B*R) % 1 # modulus 1 (i.e., % 1) take the fraction part.
print("R =", R)
d_3 = int(B*R)
print("d_3 =", d_3)

R = (B*R) % 1 # modulus 1 (i.e., % 1) take the fraction part.
print("R =", R)
d_4 = int(B*R)
print("d_4 =", d_4)

R = (B*R) % 1 # modulus 1 (i.e., % 1) take the fraction part.
print("R =", R)
```

```
d_5 = int(B*R)
print("d_5 =", d_5)
```

```
R = 0.5999999999999943
d_3 = 4
R = 0.7999999999999545
d_4 = 6
R = 0.3999999999996362
d_5 = 3
```

# 6 Binary Numbers

- Binary numbers are numbers expressed in base 2
- All modern digital computers use binary numbers internally

# 7 A fixed point real numbers

- A computer stores real number with a fixed number of binary bits
- How do we express a real number with these digits?
- One choice is to fix the number of digits in both the integer part and the fraction part. This is called the **fixed point format**
- For example, The so called **Q1.14** format uses 1 bit for the integer part and 14 bits for the fractional part, with 1 bit for the sign (+/-). The maximum value it can express is $2 - 2^{-14}$, the minimum value it can express is $-1 + 2^{-14}$. The precision is $2^{-14}$.
- Of course it cannot express $R = 0.3$ precisely,

$$(0.3)_{10} = (0.010\overline{011001100110})_2$$

  - In **Q1.14** format,

$$(0.3)_{10} \approx (0.01001100110011)_2$$

    And the error is

$$(0.3)_{10} - (0.01001100110011)_2 = (0.0000000000000000\overline{011001100110})_2$$

    This may be a small error, but it may be amplified during subsequent calculations.
- The main limitation of fixed point numbers is the very limited range of numbers it can express.

# 8 A float point number (simply called a float)

- A float point number has one sign bit $S$ (S=1 is negative), $q$ exponent bits $Q$, and $p$ precision bits $P$:

$$S \cdot Q_1 Q_2 \ldots Q_q \cdot P_1 P_2 \ldots P_{p-1}$$

  **Note that the leading bits of P is implicit (not stored, but implied from the value of Q)**
  - $Q = 0$: the leading bit is 0
  - $Q \neq 0$: the leading bit is 1

- To express positive or negative exponents, the exponent bits as an unsigned binary integer $Q$ is subtracted by a bias $b$.
- **If every bit of Q is 1**, i.e., $Q = (111\ldots1)_2 = 2^q - 1$, then **Q does not represent an exponent**, the number is either $\pm\infty$ or NaN (not a number).
- **The maximum exponent Q is** $2^q - 2$. For none zero $Q$, the precision bits are always in the form of

  $$P = 1.P_1P_2\cdots P_{p-1}$$

  The leading 1 is implicit, i.e., not stored. So we always have at most $p$ significant bits.

  $$R = \pm P \times 2^{Q-b}$$

- **The minimum exponent Q is 0**, in this case,

  $$P = 0.P_1P_2\ldots P_{p-1}$$

  ,

  $$R = \pm P \times 2^{-b+1}$$

  In this case, to get to smaller numbers, we trade off the number of significant bits.
- The smallest positive number it can express is thus $2^{-b+1-(p-1)} = 2^{-b-p+2}$. However, in this case, there is a single significant bit.
- The smallest positive number it can express that keeps $p$ significant bits is $2^{-b+1}$.
- The largest positive number it can express is $(1 - 2^{-p+1}) \times 2^{2^q-2-b}$.

## 8.1 A double precision float

- In the IEEE 754 standard: $p = 53$, $q = 11$, $b = 1023$.
- The smallest positive number it can express is $2^{-1023-53+2} = 2^{-1074} \approx 5 \times 10^{-324}$. However, in this case, there is a single significant bit.
  - Thus, in Python, $2^{-1075} = 0$! This is an example of **rounding error**, see below
- The smallest positive number to keep 53 significant bits is $2^{-1022}$
- The largest positive number it can express is $(2-2^{-52}) \times 2^{1023} \approx 1.7976931348623157 \times 10^{308}$
  - Any number larger that that is $\infty$.

### 8.1.1 Example

What is the value of the double precision float $R$ where the sign bit $S = 1$ the exponent $Q = (100\ldots0)_2 = 2^{10}$, and $P = (100\ldots0)_2 = 2^{51}$

## 8.2 Rounding error

- A rounding error is an error introduced in the finite-digit representation of real numbers. any number with more than the maximum significant bits that a float point number can represent is rounded.
  - the above example of $2^{-1075} = 0$ is a manifestation.
  - the largest integer with 52 bits of significant bits is $2^{52} - 1 = 4,503,599,627,370,495$
    - if $a/b > (2^{52} - 1)/2 \approx 2.251 \times 10^{15}$ (the /2 is to account for rounding up) then, in Python, $a - b$ has a rounding error
    - if $a/b > (2^{52} - 1)/4 \approx 1.126 \times 10^{15}$, then $b$ is rounded to 0 when computing $a - b$, i.e., $a - b = a$
  - This only depends on the relative magnitude of $a$ and $b$

[13]:
```
# another example of rounding error
a=(2.0**52-1)*1000
b = 0.2*1000
print("a-(a-b)=",a-(a-b))
print("a-b = a is", a-b == a)
print("a-a+b=",a-a+b)
print("a-a+b==a-(a-b) is",(a-a+b)==(a-(a-b)))
```

```
a-(a-b)= 0.0
a-b = a is True
a-a+b= 200.0
a-a+b==a-(a-b) is False
```

The above example shows that, due to rounding errors, in Python (actually, in all computer languages with float point numbers), the association rule may not hold for float numbers!

### 8.3 The important of Rounding errors

Rounding errors are typically small. It is curious why we worry about such a small error. Some algorithms intrinsically magnifies errors. So affter many iterations, the rounding error may be magnified to an unacceptable large value. We will see such examples in future lectures.

[ ]:

[ ]:

[ ]: