Notas de aula

Introdução à Física Computacional

Prof. Gerson – UFU – 2019

Atendimento:

- Sala 1A225
- Email: gersonjferreira@ufu.br
- Webpage: http://gjferreira.wordpress.com
- Horário: sextas-feiras 16:00 – 16:50

Exercises to guide the discussions

Factorial

Write a function my_factorial(n) to run as:

```
n = 5
x = my_factorial(n)
print("The result is", x)
```

Fibonacci

Write a function my_fibonacci(n) to run as:

```
n = 7
x = my_fibonacci(n)
print("The result is", x)
```

Bhaskara

Write a function my_bhaskara(a,b,c) to run as:

```
a = 1
b = 2
c = 3
x1, x2 = my_bhaskara(a, b, c)
print("The first root is", x1)
print("The second root is", x2)
```

Important:

→ notation to define functions

→ identation

→ type and scope of variables

Defining functions in python

General definition:

```
def name_of_the_function(a, b, c, …):
    # operations
    # …
    # foo…
    return object_to_return
```

identation

Calling the function:

```
x = name_of_the_function(4, 6, 3, ...)
```

Notice the **identation** →
It defines the beggining and end of the function

Example:

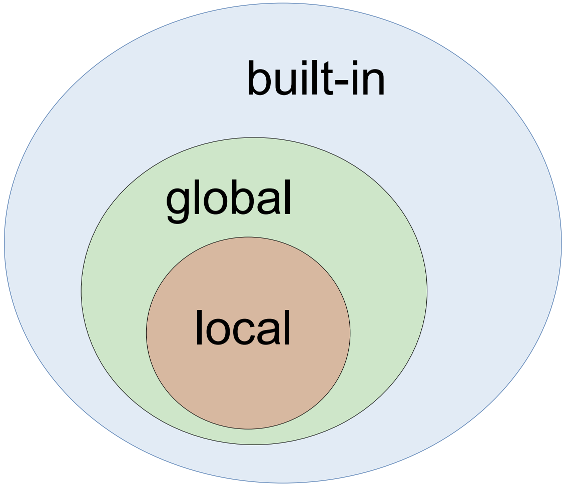
```
def average(a, b, c):
    avg = a + b + c
    avg = avg / 3

    return avg

x = average(1.5, 3.1, 2.9)
print(“The average is:”, x)
```

Compact lambda functions:

```
average = lambda a,b,c: (a+b+c)/3
x = average(1.5, 3.1, 2.9)
print(“The average is:”, x)
```

Scope of a variable

The visibility of the variable → which parts of the code can access the variable



Built-in: special names defined within the python language
Global: uppermost level of the script
Local: defined within functions/classes/packages

Example:

```
import numpy as np

def twice(x):
    y = 2*x
    return y

x = np.pi
y = 2.7
```

```
z = twice(3)
print('(x,y,z)=', x, y, z)

z = twice(y)
print('(x,y,z)=', x, y, z)

z = twice(x)
print('(x,y,z)=', x, y, z)
```

1) Try to guess the outputs

2) Which variables/constants are global or local?

Data representation

Real numbers

Example of <u>exact</u> representations:

$$123.456 = 123456 \times 10^{-3} = 1.23456 \times 10^{2}$$

significand x base $^{\text{exponent}}$

Integers

42 : what is the meaning of all this?

how does the computer store these numbers?

what are the limitations?

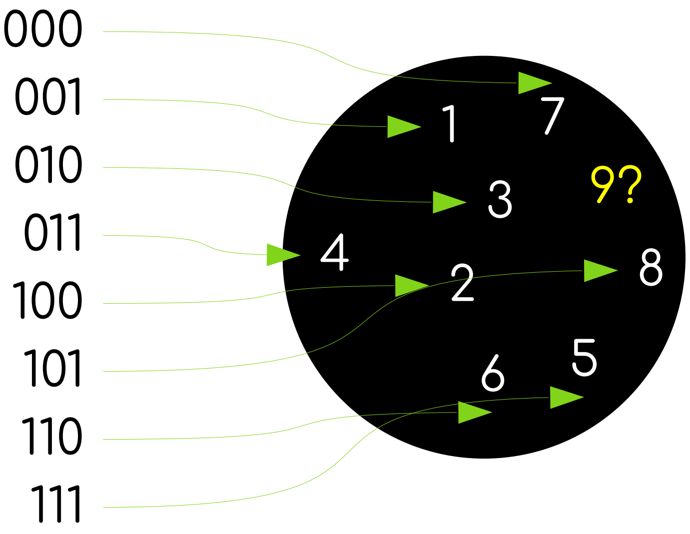
8-bits, 16-bits, 32-bits, 64-bits, 128-bits … ?

(nes, snes, ps, n64, ps2, xbox, … ;-)

Each CPU implements native n-bits operations [current PCs: 64-bits]

Example: it limits the memory allocation and data representation

A 3-bit memory controller
can only point to $2^3 = 8$ addresses



A 3-bit memory bank
can only represent 8 abstract entities

000 = apple
001 = avocado
010 = banana
011 = lemon
100 = pineapple
101 = strawberry
110 = kiwi
111 = grape

orange?

Integer representation

From the previous slide, a 3-bit computer can only represent 8 integers

3-bit binary memory

1	0	1
2	1	0

$= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$

Min: (000)b = 0
Max: (111)b = 7

Signed 32-bit integer standard

1	1	1	1	1	1	...	1	1

Sign +/-

+ 31-bits

$= (-1)^{b31} \times (b_{30} \times 2^{30} + b_{29} \times 2^{29} + \ldots b_0 \times 2^0)$

"sign-magnitude representation"

Range 9+ digits

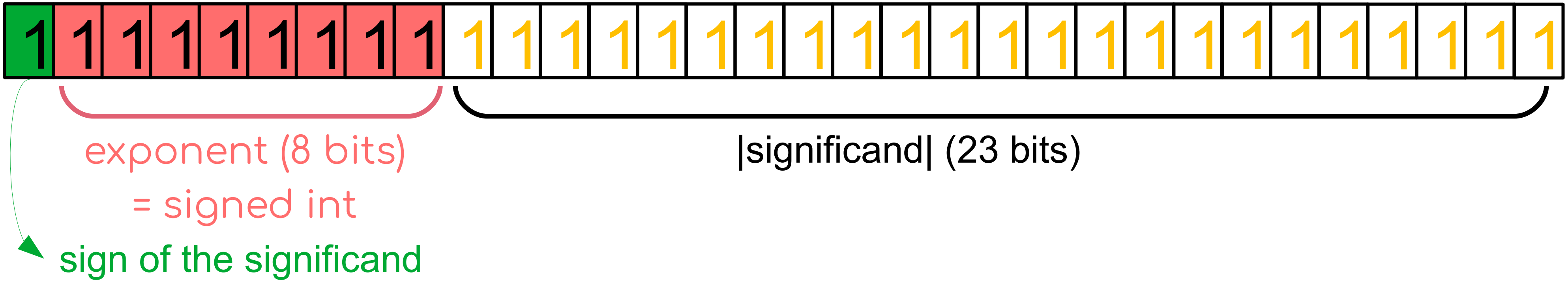
Min: $-2^{31} = -2{,}147{,}483{,}648$
Max: $+2^{31}-1 = +2{,}147{,}483{,}647$

"2's complement representation"

Real numbers : floating point

significand x base ^{exponent}

Signed 32-bit float standard



While the set of real numbers is dense / continuous...

only a limited number of entities can be represented on a finite structure

→ loss of precision

Comparing real numbers

Let us try this in Python

1) initialize the variables as

```
x = 0.1
y = 3*x
z = 0.3
```

2) check the output of the equality tests

```
z == 0.3
z == y
z == 3*0.1
```

3) What is happening? Let us print x, y, z with many digits

```
print('x = ', format(x, '0.30f'))
print('y = ', format(y, '0.30f'))
print('z = ', format(z, '0.30f'))
```

Why is 0.1 = 0.10000000000000000555111512312 6 ?

If a float in simply stored as significand x base $^{\text{exponent}}$

shouldn't we have $0.1 = 1 \times 10^{-1}$?

... but computers use base 2

Examples

→ the fraction 1/10 in base 10 is 0.1 (exact)

→ the fraction 1/3 in base 10 is 0.3 = 0.333333... ~ 0.333334

→ the fraction 1/10 in base 2 is 0.00011 = $0.0\underbrace{0011}\underbrace{0011}\underbrace{0011}\underbrace{0011}...$

An exact fraction in base 10 might be a repeating fraction in base 2

Truncation yields: $(0.0\underbrace{0011}\underbrace{0011}\underbrace{0011})_2 = (0.0999755859375)_{10}$

To convert from decimal to binary: [https://www.rapidtables.com/convert/number/binary-to-decimal.html]

Types of variables

The commands type(…) returns the type of the variable, let's try it

```
type(3)
type(3.0)

x = 2.5
type(x)

z = 1 + 2.0j
type(z)

s = 'what am I?'
type(s)
```

<class 'int'> : integer of arbitrary size

<class 'float'> : 64bits floating-point 'real' number

<class 'complex'> : floating-point complex (real) + i(imag)
→ it uses j to represent the imaginary (math)

<class 'str'> : a string

To check the size in bytes

```
import sys
x = 5.4
sys.getsizeof(x)
```

Function parameters and default values

It is possible to assign default values for parameters.
Once a function is called, if the parameter is ommited, the default value is used.

Example:
→ this function compares two floats within a default precision (eps)

```
import numpy as np

def compare_floats(x, y, eps=0.01):
    if np.abs(x-y) < eps:
        return 'equal'
    else:
        return 'different'
```

a) calling it with the default precision:

```
print( compare_floats( 3.15, 3.18 ) )
```

output: different

b) calling it with a reduced precision:

```
print( compare_floats( 3.15, 3.18, eps=0.1 ) )
```

output: equal