Python Basics!

data types, strings, indexing

CS101 Lecture #3

Administrivia

Administrivia

- Homework #1 deadline just passed.
- Solutions have been released on CodeLab.

Administrivia

- Where can you get help in this class?
  - Blackboard forum
  - Lab and office hours
  - Email me
  - *Make use of search engine!*

Administrivia

Course enrollment and CodeLab registration using zju.edu.cn account (or the intl. account) is OK.

But **Lab submissions** (to: cs101homework@intl.zju.edu.cn) have to come from your intl.zju.edu.cn account.

Administrivia

- Lab #2 tomorrow.

Quick Review & A Bit New

```
x = 10
```

How Assignment Works

```
x = 10
y = x * x
```

How Assignment Works

```
x = 10
y = x * x
x * x = y
```

```
x = 10
y = x * x
x * x = y   # error! assignment is from rhs to lhs
```

How Assignment Works

```
x = 10
y = x * x
x * x = y   # error! assignment is from rhs to lhs

x,y = y,x    # a neat trick
```

Warmup Quiz

```
x = 10
y = x + 1
y = x * y
```

What is the value of y?

  A 11
  B 100
  C 110
  D None of the above

```
x = 10
y = x + 1
y = x * y
```

What do we call x?
  A a literal
  B a variable
  C an expression
  D a statement

```
x = 10
y = x + 1
y = x * y
```

What do we call 10?

  A a literal
  B a variable
  C an expression
  D a statement

```
x = 10
y = x + 1
y = x * y
```

What do we call y = x * y?
  A a literal
  B a variable
  C an expression
  D a statement

```
x = 10
y = x + 1
y = x * y
```

What do we call x * y?
  A a literal
  B a variable
  C an expression
  D a statement

```
x = 10
y = x
x = 5
```

What is the value of y?

A 10
B 5

Data Types

Why need data type?

01001000 01000101 01001100 01001100
A computer binary code

- Computer represents different kinds of data (5, 'apple', operator $+$) in 0s and 1s

Why need data type?

01001000 01000101 01001100 01001100
A computer binary code

- Computer represents different kinds of data (5, 'apple', operator $+$) in 0s and 1s
- Different types of data are **encoded** in binary with different rules

Why need data type?

01001000 01000101 01001100 01001100
A computer binary code

- Computer represents different kinds of data (5, 'apple', operator $+$) in 0s and 1s
- Different types of data are **encoded** in binary with different rules
- What is *encoding*?

The same binary data can be interpreted in different ways based on their *data type*

Example

The same binary data can be interpreted in different ways based on their *data type*

01100111 can be the number 103, hexadecimal 67, or a letter 'g', etc.

Example

The same binary data can be interpreted in different ways based on their *data type*

01100111 can be the number 103, hexadecimal 67, or a letter 'g', etc.

In order to interpret it correctly, we need to know its data type.

What is a *data type*?

- A **data type** defines an encoding rule.

What is a *data type*?

- A **data type** defines an encoding rule.
  - i.e. how data is represented in memory by 0s and 1s.

What is a *data type*?

- A **data type** defines an encoding rule.
  - i.e. how data is represented in memory by 0s and 1s.
- It also defines the allowed operations
  - e.g. cannot do arithmetic to characters.

Numeric Data Types

Representing numbers in binary

- Binary encoding for numbers:

00000000	0	00000100	4	00001000	8
00000001	1	00000101	5	00001001	9
00000010	2	00000110	6	...	
00000011	3	00000111	7	11111111	...

Representing numbers in binary

- Binary encoding for numbers:

00000000	0	00000100	4	00001000	8
00000001	1	00000101	5	00001001	9
00000010	2	00000110	6	...	
00000011	3	00000111	7	11111111	...

- example: 01011010

Representing numbers in binary

- Binary encoding for numbers:

00000000	0	00000100	4	00001000	8
00000001	1	00000101	5	00001001	9
00000010	2	00000110	6	...	
00000011	3	00000111	7	11111111	...

- example: 01011010

  https://www.bottomupcs.com/chapter01.xhtml

- How about *Integers*?

$$..., -3, -2, -1, 0, 1, 2, 3, ...$$

Integers (int), $\mathbb{Z}$

- How about *Integers*?

$$..., -3, -2, -1, 0, 1, 2, 3, ...$$

- Negative numbers

- How about *Integers*?

$$..., -3, -2, -1, 0, 1, 2, 3, ...$$

- Negative numbers
  - Use the leftmost bit as **sign bit**.

Integers (int), $\mathbb{Z}$

- How about *Integers*?

$$..., -3, -2, -1, 0, 1, 2, 3, ...$$

- Negative numbers
  - Use the leftmost bit as **sign bit**.
  - 0 for positive; 1 for negative

Integers (int), $\mathbb{Z}$

- How about *Integers*?

$$..., -3, -2, -1, 0, 1, 2, 3, ...$$

- Negative numbers
  - Use the leftmost bit as **sign bit**.
  - 0 for positive; 1 for negative
  - the rest of the bits representing magnitude

Integers (int), $\mathbb{Z}$

- How about *Integers*?

$$..., -3, -2, -1, 0, 1, 2, 3, ...$$

- Negative numbers
  - Use the leftmost bit as **sign bit**.
  - 0 for positive; 1 for negative
  - the rest of the bits representing magnitude
- What are the limits of a 8-bit integer representation?

Integers (int), $\mathbb{Z}$

- How about *Integers*?

$$..., -3, -2, -1, 0, 1, 2, 3, ...$$

- Negative numbers
  - Use the leftmost bit as **sign bit**.
  - 0 for positive; 1 for negative
  - the rest of the bits representing magnitude
- What are the limits of a 8-bit integer representation?

$$-128...127$$

History

- Old version python **int** are 32 bits long (in the range of -2^{31} to $2^{31} - 1$)
- That's -2147483648 to 2147483647
- values too big: *overflow*
- values too small: *underflow*

History

- Python has another integer type: **long**
- Represents with no restrictions on size (no overflow/underflow)

History

- Python has another integer type: **long**
- Represents with no restrictions on size (no overflow/underflow)
- Since v2.2, python converts int overflow to a **long**

History

- Python has another integer type: **long**
- Represents with no restrictions on size (no overflow/underflow)
- Since v2.2, python converts int overflow to a **long**
- newer Python versions promises there is no distinction between **int** and **long**

History

- Python has another integer type: **long**
- Represents with no restrictions on size (no overflow/underflow)
- Since v2.2, python converts int overflow to a **long**
- newer Python versions promises there is no distinction between **int** and **long**
- Don't get spoiled by this (many languages still have clear integer types and limits).

https://en.wikipedia.org/wiki/Integer_(computer_science)

- Evaluating an expression of integers will generally result in an integer answer
  - 3 + 5

- Evaluating an expression of integers will generally result in an integer answer
  - 3 + 5
  - EXCEPTION: DIVISION!

Integer operations

- Evaluating an expression of integers will generally result in an integer answer
  - `3 + 5`
  - EXCEPTION: DIVISION!
  - `3 / 4 → 0.75`

Integer operations

- Evaluating an expression of integers will generally result in an integer answer
  - 3 + 5
  - EXCEPTION: DIVISION!
  - 3 / 4 → 0.75
  - 3 // 4 → 0 (floor division)

Integer operations

- Evaluating an expression of integers will generally result in an integer answer
  - 3 + 5
  - EXCEPTION: DIVISION!
  - 3 / 4 → 0.75
  - 3 // 4 → 0 (floor division)
  - 4 / 2 → ??

- Floating-point numbers include a fractional part. (Anything with a decimal point—2.4, 3.0.)

Floating-point numbers, $\mathbb{R}$

- Floating-point numbers include a fractional part.
  (Anything with a decimal point—2.4, 3.0.)
- What's different?

Floating-point numbers, $\mathbb{R}$

- Floating-point numbers include a fractional part. (Anything with a decimal point—`2.4`, `3.0`.)
- What's different?
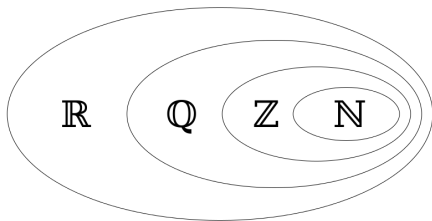 - represents *Real numbers* (integers, fractional, and π, *e*, sqrt(2), etc.)

Floating-point numbers, $\mathbb{R}$

- Floating-point numbers include a fractional part. (Anything with a decimal point—2.4, 3.0.)
- What's different?
  - represents *Real numbers* (integers, fractional, and π, *e*, sqrt(2), etc.)
  - up to a precision and limit (maximum and minimum)

Real numbers (R) include the rational (Q), which include the integers (Z), which include the natural numbers (N).

- Evaluating an expression of floating-point values will result in a floating-point answer.

Floating-point operations

- Evaluating an expression of floating-point values will result in a floating-point answer.
  - `3.0 + 5.5` → `8.5`

Floating-point operations

- Evaluating an expression of floating-point values will result in a floating-point answer.
  - 3.0 + 5.5 → 8.5
  - 3.0 + 5.0 → 8.0

Floating-point operations

- Evaluating an expression of floating-point values will result in a floating-point answer.
  - `3.0 + 5.5` → `8.5`
  - `3.0 + 5.0` → `8.0`
  - `3 + 5.5` → `?` (what happens here?)

Floating-point operations
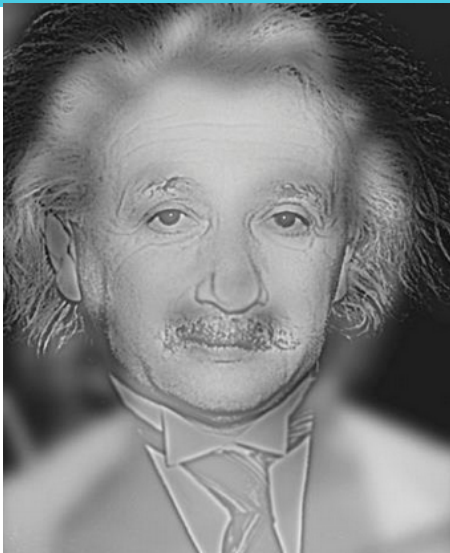
- Evaluating an expression of floating-point values will result in a floating-point answer.
  - 3.0 + 5.5 → 8.5
  - 3.0 + 5.0 → 8.0
  - 3 + 5.5 → ? (what happens here?)
- Engineers and scientists need to think carefully about data type, precision, and type *conversion*.

Einstein or Monroe?

String Data Type

ASCII encoding table

```
000     (nul)    016 ▶ (dle)    032 sp    048 0    064 @    080 P    096 `    112 p
001 ☺ (soh)    017 ◀ (dc1)    033 !    049 1    065 A    081 Q    097 a    113 q
002 ☻ (stx)    018 ↕ (dc2)    034 "    050 2    066 B    082 R    098 b    114 r
003 ♥ (etx)    019 ‼ (dc3)    035 #    051 3    067 C    083 S    099 c    115 s
004 ♦ (eot)    020 ¶ (dc4)    036 $    052 4    068 D    084 T    100 d    116 t
005 ♣ (enq)    021 § (nak)    037 %    053 5    069 E    085 U    101 e    117 u
006 ♠ (ack)    022 ▬ (syn)    038 &    054 6    070 F    086 V    102 f    118 v
007 • (bel)    023 ↨ (etb)    039 '    055 7    071 G    087 W    103 g    119 w
008 ◘ (bs)     024 ↑ (can)    040 (    056 8    072 H    088 X    104 h    120 x
009    (tab)    025 ↓ (em)     041 )    057 9    073 I    089 Y    105 i    121 y
010    (lf)     026    (eof)    042 *    058 :    074 J    090 Z    106 j    122 z
011 ♂ (vt)     027 ← (esc)    043 +    059 ;    075 K    091 [    107 k    123 {
012 ♀ (np)     028 ∟ (fs)     044 ,    060 <    076 L    092 \    108 l    124 |
013    (cr)     029 ↔ (gs)     045 -    061 =    077 M    093 ]    109 m    125 }
014 ♫ (so)     030 ▲ (rs)     046 .    062 >    078 N    094 ^    110 n    126 ~
015 ☼ (si)     031 ▼ (us)     047 /    063 ?    079 O    095 _    111 o    127 ⌂
```

ASCII encoding table

000 (nul)	016 ▶ (dle)	032 sp	048 0̇	064 @	080 P	096 `	112 p
001 ☺ (soh)	017 ◀ (dc1)	033 !	049 1	065 A	081 Q	097 a	113 q
002 ☻ (stx)	018 ↕ (dc2)	034 "	050 2	066 B	082 R	098 b	114 r
003 ♥ (etx)	019 ‼ (dc3)	035 #	051 3	067 C	083 S	099 c	115 s
004 ♦ (eot)	020 ¶ (dc4)	036 $	052 4	068 D	084 T	100 d	116 t
005 ♣ (enq)	021 § (nak)	037 %	053 5	069 E	085 U	101 e	117 u
006 ♠ (ack)	022 ▬ (syn)	038 &	054 6	070 F	086 V	102 f	118 v
007 • (bel)	023 ↨ (etb)	039 '	055 7	071 G	087 W	103 g	119 w
008 ◘ (bs)	024 ↑ (can)	040 (	056 8	072 H	088 X	104 h	120 x
009 (tab)	025 ↓ (em)	041)	057 9	073 I	089 Y	105 i	121 y
010 (lf)	026 (eof)	042 *	058 :	074 J	090 Z	106 j	122 z
011 ♂ (vt)	027 ← (esc)	043 +	059 ;	075 K	091 [	107 k	123 {
012 ♀ (np)	028 ∟ (fs)	044 ,	060 <	076 L	092 \	108 l	124	
013 (cr)	029 ↔ (gs)	045 -	061 =	077 M	093]	109 m	125 }
014 ♫ (so)	030 ▲ (rs)	046 .	062 >	078 N	094 ^	110 n	126 ~
015 ☼ (si)	031 ▼ (us)	047 /	063 ?	079 O	095 _	111 o	127 ⌂

The table provides an *encoding* scheme from symbols to numbers

72 69 76 76 79 = H E L L O

- H E L L O $= 72\ 69\ 76\ 76\ 79$

How to store text on computer?

- `H E L L O` $= 72\ 69\ 76\ 76\ 79$
- Each symbol is stored individually, one byte long:

How to store text on computer?

- H E L L O = 72 69 76 76 79
- Each symbol is stored individually, one byte long:

  72   01001000
  69   01000101
  76   01001100
  76   01001100
  79   01001111

How to store text on computer?

- H E L L O = 72 69 76 76 79
- Each symbol is stored individually, one byte long:

```
72  01001000
69  01000101
76  01001100
76  01001100
79  01001111
```
`'HELLO'` : 01001000 01000101 01001100 01001100 01001111

- What's the size of a plain txt file with 1000 english words?