Basic Data Types in Python

Prof. Pai H. Chou
National Tsing Hua University

Outline

- Characters
  - ASCII code, Unicode
- Numbers
  - integers, floating point, complex
  - Operators: Comparison, Bitwise, Shifting
- Boolean
  - Truth interpretation of other types
- Comparisons

Characters

- Basic unit of text display
- ASCII character set
  - text characters, control characters
  - use of ord() and chr() functions
- Unicode
  - extended characters
  - example by list comprehension

ASCII Character set

- American Standard Code for Information Interchange
  - basis for virtually all programming languages

Code	Char
32	SPACE
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/

Code	Char
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?

Code	Char
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O

Code	Char
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_

Code	Char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o

Code	Char
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~
127	DEL

Try built-in functions

- `ord(ch)`: look up a character's code
- `chr(co)`: map from code to character

```
>>> ord('A')
65
>>> chr(65)
'A'
>>> s = ''
>>> for i in range(65, 91):
...     s += chr(i)
...
>>> s
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>>
```

- note: end='' suppresses newline when printing

Control Characters in ASCII: 0..31

- Purpose: for text display or printer control

Code	Char
0	NUL (null)
1	SOH (start of heading)
2	STX (start of text)
3	ETX (end of text)
4	EOT (end of transmission)
5	ENQ (enquiry)
6	ACK (acknowledge)
7	BEL (bell)
8	BS (backspace)
9	TAB (horizontal tab)
10	LF (newline)
11	VT (vertical tab)
12	FF (form feed)
13	CR (carriage return)
14	SO (shift out)
15	SI (shift in)

Code	Char
16	DLE (data link escape)
17	DC1 (device control 1)
18	DC2 (device control 2)
19	DC3 (device control 3)
20	DC4 (device control 4)
21	NAK (negative acnowledge)
22	SYN (synchronous idle)
23	ETB (end of trans. block)
24	CAN (cancel)
25	EM (end of medium)
26	SUB (substitute)
27	ESC (escape)
28	FS (file separator)
29	GS (group separator)
30	RS (record separator)
31	US (unit separator)

Control characters

- Common assumption: terminal or printer
  - idea: move the cursor, overwrite
- Example
  - BEL => beeps!

7	BEL (bell)	`'\x07'`
8	BS (backspace)	`'\b'`
9	TAB (horizontal tab)	`'\t'`
10	LF (newline)	`'\n'`
13	CR (carriage return)	`'\r'`

  - BS => backspace (move cursor left by one position)
  - TAB => move cursor to next 8x column
  - LF => move cursor to beginning of next line
  - CR => move cursor to beginning of this line

Example: text clock

```
$
```

- repeat forever (until the user kills it with Ctrl-C)
  - get current time (hour, minute, second)
  - move cursor to leftmost column, print current time
  - wait for one second

Source code for clock.py

```
import time

while True:
    t = time.localtime()
    print('\r%02d:%02d:%02d' % (t.tm_hour, t.tm_min, t.tm_sec), end='')
    time.sleep(1)
```

- `time` module provides
  - `localtime()` => get the time right now
  - `sleep(t)` => wait for `t` sec before continuing.
- `'\r'` moves cursor to beginning of line
- can be extended to beep by printing `chr(8)` or `'\x08'`

Extended Characters: Unicode

- Beyond the 0-127 ASCII code
- Python3 handles unicode in source code
  - identifiers, strings, comments, `chr()`, `ord()`

```
>>> [ord('♠'), ord('♣'), ord('♥'), ord('♦')]
[9824, 9827, 9829, 9830]
>>> [chr(9824), chr(9827), chr(9829), chr(9830)]
['♠', '♣', '♥', '♦']
>>> ord('一')  # handles Hanzi/Kanji
19968
>>> ord('ㄅ')  # handle bopomofo
12549
```

List Comprehension for generating a range of characters

- Hanzi and Kanji are ordered by radical

```
>>> ord('一') # "one" in Chinese character
19968
>>> [chr(i) for i in range(19968, 19978)] # list comprehension
['一', '丁', '丂', '七', '丄', '丅', '丆', '万', '丈', '三']
>>> ord('虫')
34411
>>> [chr(i) for i in range(34411,34421)] # like dictionary order
['虫', '虬', '虮', '虰', '虱', '虹', '虺', '虻', '虼', '蚃']
>>> chr(12549)
'ㄅ'
>>> tuple(chr(i) for i in range(12549,12586))
('ㄅ', 'ㄆ', 'ㄇ', 'ㄈ', 'ㄉ', 'ㄊ', 'ㄋ', 'ㄌ', 'ㄍ', 'ㄎ', 'ㄏ',
'ㄐ', 'ㄑ', 'ㄒ', 'ㄓ', 'ㄔ', 'ㄕ', 'ㄖ', 'ㄗ', 'ㄘ', 'ㄙ', 'ㄚ',
'ㄛ', 'ㄜ', 'ㄝ', 'ㄞ', 'ㄟ', 'ㄠ', 'ㄡ', 'ㄢ', 'ㄣ', 'ㄤ', 'ㄥ',
'ㄦ', 'ㄧ', 'ㄨ', 'ㄩ')
```

Use of Unicode in program

- Unicode can be valid identifiers
  - `名字 = input('請問您的大名是? ')`
  - `# 註解也可以使用 Unicode`
- Permissible but not encouraged in general
  - String literal: better to externalize them
  - You may want to share source code with others
    - they may not understand your local language
- Better to write all in ASCII subset in English

Summary: Characters

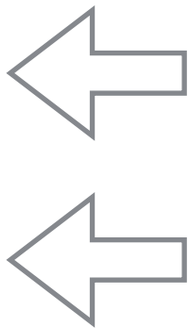
- Character: basic unit of text display
  - Internally, each character is represented as a code
- Code standards
  - ASCII: 0-31 = control, 32-127 = text symbols
  - Unicode: includes ASCII and extended characters
- Python functions
  - chr(c): maps code c to character
  - ord(ch) maps character ch to code
  - list comprehension is useful for working with code range

Literals vs. Expressions

- Literal: a notation for a constant value (according to Python syntax rule)
  - character literals: `'hello'`, `'world'`, `"that's good"`, `"""triple quotes"""`, `'''and more'''`
  - integer literals: `12`, `-3`, `0x1a2b`, `0o43`, `0b1010`
  - floating point literals: `12.3`, `-3.45e5`
- Expressions: anything that has a value
  - literal, variable, return value of function call, expressions composed by operator
  - `x`, `x+2*y`, `f(x)`, ...

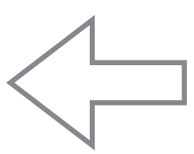
Value of literals

- String literal: content inside the quotes
  - string value does not include the quotes!
  - so, `print('hello')` prints
    `hello`
    not
    `'hello'`

    value of the string literal 'hello'

    the string literal
- In interactive mode, python echos literals
  - ```
    >>> s = 'hello'
    >>> s
    'hello'
    ```

    python interactive mode displays the value in *string literal* format!
    ```

Literal vs. Variable

- `print("hello")` prints `hello`
- `print(hello)` prints value of variable
  - the output depends on what value is assigned to hello
  - if hello has not been assigned a value, then it is an error!
  - if `hello` is assigned string, it prints that string's value
  - if `hello` is assigned another type, Python converts it to string before printing it (e.g., integer to string)

Numbers

- Number vs. numeral vs. literals
- integers
  - can be arbitrarily large (unlike other languages)
  - can be expressed in terms of different bases
- floating point
  - limited in size by hardware support
  - can be written in scientific notation
- complex numbers
  - two dimensional: real and imaginary components

Number notations

- `n = 123; print(n)`
  - variable n gets integer value of literal 123;
  - `print(n)` prints characters '1', '2', '3'
    i.e., `chr(49), chr(50), chr(51)`

- Three concepts
  - number: the numeric quantity
  - numeral: a (general) notation for numbers
  - literal: a (Python) notation for numbers

Code	Char
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?

Numeral: a notation for a number

- Fingers: 
- Arabic: 1, 2, 3, 4, 5, ...10, 11, 12, ..
  - Assumption: base-10, positional
- Roman numeral: I, II, III, IV, V,... X, XI, XII
- Chinese: 一,二,三,四,五,...十,十一,十二
- Bank: 壹,貳,叁,肆,伍,...拾,捨壹,拾貳
- => different ways to denote numbers

integer literals in Python

- Binary: 1011_0101_1101_0101
  - in Python, `0b1011_0101_1101_0101` (leading `0b` => binary)
- Octal: make groups of 3 bits
  - 1_011_010_111_010_101 (base 2)
    = 1 3 2 7 2 5 (base 8) = 132725_8
  - in Python syntax, `0o132725` (leading `0o` => octal)
- Hex: make groups of 4 bits
  - 1011_0101_1101_0101 (base 2)
    = B 5 D 5 (base 16) = $B5D5_{16}$
  - in Python syntax, `0xB5D5` (leading `0x` => hex)

Python functions for converting int to hex/octal literal

- Converts integer (number) to literal in a given base
  - `oct(`*n*`)` converts integer to octal literal string
  - `hex(`*n*`)` converts integer to hex literal string
  - `bin(`*n*`)` converts integer to binary literal string
  - `str(`*n*`)` converts integer to decimal literal string
- `int(s)` converts string of diff. formats to int

```
>>> 0o132725       # octal
46549              # python renders it as decimal by default
>>> hex(46549)    # convert to its hex string
'0xb5d5'
>>> hex(0o132725)
'0xb5d5'
>>> oct(46549)
'0o132725'
```

Integers in Python compared to other languages

Language	Python	C	Java
Size	unlimited	32 or 64 bits	32 or 64 bits
digit separator	_	N/A	maybe _
bases supported	decimal, binary, octal, hex	decimal, octal, hex	decimal, binary (7.1), octal, hex

Summary: binary number representations

- Machine: binary (base 2)
  - each binary digit (bit) is internally 0 or 1
  - need conversion to show in decimal (base 10)
- Notation for binary
  - binary: base 2, `0b` prefix, digits `0..1`
  - octal: base 8, `0o` prefix, digits `0..7`
  - hex: base 16, `0x` prefix, digits `0..9,A..F`

Bitwise operators

- binary operators
  - **&**: bitwise AND
  - **|**: bitwise OR
  - **^**: bitwise XOR (exclusive-OR)
- unary operator
  - **~**: bitwise NOT (or bit-invert)

Bitwise AND operator &

- Truth tables

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

```
  0 1 1 0 1 0
& 1 0 1 0 0 1
-------------
  0 0 1 0 0 0
```

- Each int is a "bit vector" of multiple bits

```
>>> a = 0b_011_010
>>> b = 0b_101_001
>>> bin(a & b)      # bitwise AND, get binary 001000
'0b1000'
```

Bitwise OR operator |

- Truth tables

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

```
  0 1 1 0 1 0
| 1 0 1 0 0 1
-------------
  1 1 1 0 1 1
```

- Each int is a "bit vector" of multiple bits

```
>>> a = 0b_011_010
>>> b = 0b_101_001
>>> bin(a & b)      # bitwise AND, get binary 001000
'0b1000'
>>> bin(a | b)      # bitwise OR, get binary 111011
'0b111011'
```

Bitwise XOR operator ^

- Truth tables

a	b	a ^ b
0	0	0
0	1	**1**
1	0	**1**
1	1	**0**

```
  0 1 1 0 1 0
^ 1 0 1 0 0 1
-------------
  1 1 0 0 1 1
```

- Each int is a "bit vector" of multiple bits

```
>>> a = 0b_011_010
>>> b = 0b_101_001
>>> bin(a & b)      # bitwise AND, get binary 001000
'0b1000'
>>> bin(a | b)      # bitwise OR, get binary 001000
'0b111011'
>>> bin(a ^ b)      # bitwise XOR, get binary 110011
'0b110011'
```

Summary: bitwise operators

- AND:
  - 1 if both bits are 1
- OR:
  - 1 if either or both bits 1
- XOR:
  - 1 if bits are different

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	**1**

```
  0 1 1 0 1 0
& 1 0 1 0 0 1
-------------
  0 0 1 0 0 0
```

a	b	a | b
0	0	0
0	1	**1**
1	0	**1**
1	1	**1**

```
  0 1 1 0 1 0
| 1 0 1 0 0 1
-------------
  1 1 1 0 1 1
```

a	b	a ^ b
0	0	0
0	1	**1**
1	0	**1**
1	1	**0**

```
  0 1 1 0 1 0
^ 1 0 1 0 0 1
-------------
  1 1 0 0 1 1
```

Negative integer representation and Bit Shifting

- signed integers:
  - 2's complement representation
- shifting bits in integers
  - left shift: always fill in 0's on the right
  - right-shift: fill in sign bit on the left

Signed number representation: 2's complement

- think odometer
  - to add, turn in increasing direction
  - to subtract, turn down in decreasing direction

decimal	binary	hex
2	0000_0010	0x02
1	0000_0001	0x01
0	0000_0000	0x00
-1	1111_1111	0xFF
-2	1111_1110	0xFE
-3	1111_1101	0xFD

- To negate a number, flip all bits, then + 1

Bit inversion operator ~

- ~*n*
  - flips all bits in *n*

a	~a
0	1
1	0

```
  ^ 0 1 0 0 1 0
-------------
    1 0 1 1 0 1
```

```
>>> a = 0b_010_010
>>> a                  # display as decimal
18
>>> ~a                 # flip bits - gets interpreted as negated - 1
-19
```

- How? 2's complement
  - because –18 = (~18 + 1) by 2's complement,
    => ~18 = –18 – 1 = –19.

Viewing bits in negative quantity

- Python tracks negative signs for integers
  - why? because it needs to track integer size
- To view, must limit to a given bit width
  - simplest way is to **&** it with a bit-mask

```
>>> a = 0b_010_010
>>> a                  # display as decimal
18
>>> ~a                 # flip bits - gets interpreted as negated - 1
-19
>>> bin(a)             # as binary
'0b010010'
>>> bin(~a & 0b_111_111)  # as 6-bit binary quantity
'0b101101'
```

Left-shift operator <<

- *x* << *y*: shift bits in *x* by *y* positions to left
  - fill in zero's on the right
- Effect is multiply by 2 per shifted position
  - << 3 is effectively multiplying by 2**3 = 8

```
>>> 8 << 2              # same as 0b1000 << 2
32
>>> -1 << 3
-8
>>> bin(-8 & 0xff)  # always fills in 0s on right on shifting
'0b11111000'
```

Right-shift operator >>

- *x* >> *y*: shift bits in x by y positions to right
  - if negative, high bits are filled with 1's!
  - if nonnegative, high bits are filled with 0's
- Effect is like divide by 2,
  - but shifting -1 to right will still be -1 (fill in 1's)

```
>>> 8 >> 1
4
>>> -8 >> 1
-4
>>> -1 >> 2          # -1 is all 1111..; fills in 1's (sign bit)
-1
>>> bin(-8 & 0xff)  # always fills in 0s on right on shifting
'0b11111000'
```

Summary: negative integer and bit shifting

- ~: inversion (bit-complement) operator
  - turn all 0 bits into 1, all 1 bits into 0
- 2's complement representation
  - to negate an integer, invert all bits and + 1
- bit-shift operators << and >>
  - allows bits to be shifted by a number of positions
  - left shift: fill in 0s;
    right-shift: fill in sign bit.

built-in math functions

- `abs(`*n*`)`: absolute value
  - for complex numbers, sqrt(real^2+imag^2)
- `divmod(`*a*`, `*b*`)`: (quotient, remainder tuple)
  - `divmod(10, 3)` yields `(3, 1)`
- `pow(`*a*`, `*b*`)`: a^b
  - same as `a ** b`
- `round(`*f*`)`: rounds to nearest even integer ("Banker's")
  - `round(-2.5)` and `round(-1.5)` both yield -2
  - `round(-3.5)` and `round(-4.5)` both yield -4

Complex numbers

- a "number" with real and imaginary components
  - where imaginary component j = $\sqrt{-1}$

```
>>> n = 2+3j
>>> n.real
2
>>> n.imag
3
>>> abs(n)
3.605551275463989
```

```
>>> n
(2+3j)
>>> n.conjugate()
(2-3j)
>>> -n
(-2-3j)
```

- Complex conjugate: negate the imaginary component

Library modules related to math

- library must be imported explicitly
- Standard modules that can be imported
  - **`math`**: `sin`, `cos`, `pi`, `e`
  - **`random`**: random number generator
  - **`cmath`**: complex numbers
- 3rd-party modules that can be installed
  - **`numpy`**: arrays, matrices, advanced operators

Boolean

- `bool` class
  - Possible values: `False`, `True`
- Boolean interpretation of other types
  - numbers: (integers, floats, complex)
  - strings
  - collection type: (dictionary, set, list, tuple)
  - other objects vs. `None`
- Boolean Operators
  - Short circuit evaluation
  - Negation operator

Truth values ("boolean")

- `True`, `False` are two keywords
  - result of comparison always take on one of these values
- any zero or empty value tests to `False`
  - list, string, dictionary, set, tuple, `None`
- All other values evaluate to `True`
- `bool`(*expression*) to convert to `True`/`False`
  -

```
>>> bool([])
False
>>> bool(123)
True
```

Operators on boolean values: A `and` B, A `or` B

- **`and`**: True if <u>both</u> A and B are True
- **`or`**: True if <u>either</u> or <u>both</u> A, B True

A	B	A **`and`** B
False	False	False
False	True	False
True	False	False
True	True	True

A	B	A **`or`** B
False	False	False
False	True	True
True	False	True
True	True	True

Short-circuit evaluation: and

- if A false, return A; # skip evaluating B
  else B
  - i.e., if A False, outcome can't possibly be true
    => no need to evaluate B!
  - but if A True, then evaluate B to determine outcome
- can run faster if A false

A	B	A **and** B	same as
False	False	False	A
False	True	False	
True	False	False	B
True	True	True	

```
>>> 'Hello' and []  # A true => return B, which is []
[]
>>> 2-2 and 4+8     # 2-2=0 is false => returns 0; skip 4+8
0
```

Short-circuit evaluation: `or`

- if A true, return A; # skip evaluating B
  else B
  - i.e., if A true, outcome can't possibly be false
    => no need to evaluate B!
  - but if A false, then evaluate B
    to determine outcome
- can run faster if A true

A	B	A `or` B	same as
False	False	False	B
False	True	True	
True	False	True	A
True	True	True	

```
>>> 'Hello' or []   # A true => return A, which is 'Hello'
'Hello'
>>> 2-2 or 4+8     # 2-2=0 is false => evaluate B, 4+8 => 12
12
```

Why Short-Circuit Evaluation?

- Can be faster and avoids exceptional condition
- e.g.: return the 0^{th} element of a list if it is an int; otherwise (list empty or not an int) then return 0.

```
def getFirstInt(L):
    if len(L) > 0 and type(L[0]) == int:
        return L[0]
    else:
        return 0
```

- Without short-circuit evaluation, type(L[0]) could cause an exception when L is empty!

`not`: logical negation

- `not` true evaluates to `False`
- `not` false evaluates to `True`
- Result is always boolean type (`bool`)

```
>>> X = [1, 2, 3]
>>> Y = ''
>>> not X
False
>>> not Y
True
```

Summary: boolean values and operators

- `True`, `False` are values of `bool` type
- Other types (int, float, list, set, dict, ...) can have boolean interpretation
  - Zero number, empty collection, or None => false
  - Non-zero, non-empty, objects => true
- Short-circuit evaluation for `and`, `or`
  - Skip evaluating right-hand-side expression if the left-hand-side value is same as result of operator

Comparison of Simple Values

- Numbers
  - int, float
- Characters and Strings
  - lexicographical order
- Boolean
  - False, True
- Complex: not comparable

Comparison operators

operator	meaning	operator	meaning
`<`	less than	`<=`	less than or equal
`>`	greater than	`>=`	greater than or equal
`==`	equal	`!=`	not equal

- Important: `==` vs. `=`
  - `==` is equality comparison, yields `True` or `False`
  - `=` is assignment operator
- Works on different types
  - `int`, `float`, `complex` (equality/inequality only)
  - `str`, `list` (lexicographical order)
  - `set`: subset / superset comparison

Comparison examples

```
>>> a = 3
>>> b = 2
>>> c = 1.02
>>> a > b
True
>>> b < c
False
>>> a > b > c      # same as (a > b) and (b > c)
True
>>> a > b < c      # same as (a > b) and (b < c)
False
>>> 5+4j >= 2-3j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>=' not supported between instances of 'complex' and
'complex'
>>> 0 < '0'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```

Character Ordering

- Each character has its own integer code

works for ASCII code

```
>>> ord('A')
65
>>> chr(65)
'A'
>>> ord('a')
97
```

works for unicode also

```
>>> ord('大')
22823
>>> chr(22823)
'大'
```

- Characters can be ordered by their code
  - `'0' < '1' < '2' < ... 'A' < 'B' < 'C' < ... 'X' < 'Y' < 'Z' < ... 'a' < 'b' < 'c' < ... 'x' < 'y' < 'z'`
- Comparison is case sensitive!

Lexicographical order of strings

- "dictionary order", but case sensitive
  - '' < 'A' < 'AA' < 'AAB' < 'AB' < 'ABA' ... 'AC' < ..
- Criteria:
  - compare character-by-character from beginning
  - if same prefix, then use suffix for tie-breaker
  - empty string is ordered before all other strings

Summary:
Comparison of Simple Values

- Numbers (integers and floats) can be compared
- Characters and strings can be compared - in *lexicographical* order
- `bool()` converts value to `True`/`False`
- Complex numbers cannot be compared