

Some Python

Bitwise operations in Python

Besides Boolean operators (and, or, not), we have **bitwise** operators: &, |, ^, ~, <<, >>

```
>>> 12 & 5  
4
```

```
>>> 0b1100 & 0b0101  
4
```

```
>>> bin(0b1100 & 0b0101)  
'0b100'
```

Bitwise AND example

```
      .      .  
-----  
a    0b1100  
b    0b0101  
a & b 0b0100
```

Bitwise OR example

```
      .      .  
-----  
      a  0b1100  
      b  0b0101  
a | b  0b1101
```

Bitwise XOR example

```
      .      .  
-----  
a    0b1100  
b    0b0101  
a ^ b 0b1001
```

Shift left

		.		.

	a	0b0000100		
a << 3		0b0100000		

Note: $\ll n$ is equivalent to $\times 2^n$

Shift right

```
      .           .  
-----  
      a  0b0010000  
a >> 3  0b0000010
```

Note: $\gg n$ is equivalent to (truncated) division by 2^n

Python formatted string literals

Formatted string literals look like string literals, but are prepended with an f:

```
f'Hello'
```

They allow us to:

- build a string from python expressions
- specify how to format those expressions

Syntax:

```
f'raw_string {python_expression : format} ...'
```

- `raw_string`: anything not between “{” and “}” is a raw string literal
- between “{” and “}”, we specify a formatted expression
 - `python_expression`: most python expressions are allowed here
must not be ambiguous, e.g. no “:”
 - `format`: specifies how to convert the corresponding expression to a string
- the f-string is *immediately* evaluated into a string:

```
>>> x = 3
>>> f'Hello {x}'
'Hello 3'
>>> type(f'Hello {x}')
<class 'str'>
```

Format:

- < for left align, > for right-align
- + include sign even for positive numbers
- (space) empty space for positive numbers
- b binary, d decimal (default), x hexadecimal
- f floating point number (fixed-point notation)
- 20 (or other number) specify the minimum width of the conversion result
- .10 (or other number) specify the number of digits after the decimal dot

Examples:

```
>>> f'pi ≈ |||{math.pi:+6.2f}|||'  
'pi ≈ ||| +3.14|||'  
>>> f'pi ≈ |||{math.pi:<+6.2f}|||'  
'pi ≈ |||+3.14 |||'
```

Documentation:

- > [f-strings](#)
- > [format specification](#)

String methods

- `str.find(substr)`: Return the lowest index in the `str` where substring `substr` is found. Return -1 if `substr` is not found.
- `str.replace(old, new)`: Return a copy of `str` with all occurrences of substring `old` replaced by `new`.
- `str.strip(chars)`: Return a copy of the string with the leading and trailing characters removed. The `chars` argument is a string specifying the set of characters to be removed. If omitted or `None`, the `chars` argument defaults to removing whitespace.

- `str.split(sep=None, maxsplit=-1)`: Return a list of the words in the string, using `sep` as the delimiter string. If `sep` is `None`, runs of consecutive whitespace are regarded as a single separator (the result will contain no empty strings). If `maxsplit` is given, at most `maxsplit` splits are done.
- `str.join(iterable)`: Return the concatenation of the strings `iterable`. The separator between elements is `str`.

> string methods

Conditional expressions

Syntax:

```
x if C else y
```

Example:

```
>>> x = 5  
>>> 'big' if x > 10 else 'small'  
'small'
```

Only the appropriate value is evaluated

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero>
```

Only the appropriate value is evaluated (example)

```
>>> import math
>>>
>>> x = 4
>>> 1 / x if x != 0 else math.inf
```

```
0.25
```

```
>>> x = 0
>>> 1 / x if x != 0 else math.inf
```

```
inf
```

> [documentation](#)

List comprehensions

List comprehension syntax:

```
[ expression for variable in iterable ]
```

- iterates over the values in `iterable`
- at each iteration, assign value to `variable`
- evaluate `expression` (may refer to `variable`)
- result becomes one list entry

```
>>> [i * 2 for i in range(8)]
```

```
[0, 2, 4, 6, 8, 10, 12, 14]
```

```
>>> [str(i) for i in range(8)]
```

```
['0', '1', '2', '3', '4', '5', '6', '7']
```

```
>>> [f' {i:02d} ' for i in range(8)]
```

```
['00', '01', '02', '03', '04', '05', '06', '07']
```

```
>>> [f' {i:03b} ' for i in range(8)]
```

```
['000', '001', '010', '011', '100', '101', '110', '111']
```

Dictionary comprehensions

Dictionary comprehension syntax:

```
{ key_expr : val_expr for variable in iterable }
```

Same as list comprehension, but for dict.

Example:

```
>>> { i : f' {i:02b}' for i in range(4) }  
{0: '00', 1: '01', 2: '10', 3: '11'}
```

Generator expressions

Generator expression syntax:

(same as list comprehension, but with parentheses)

```
( expression for variable in iterable )
```

Same as list comprehension, but creates a generator (iterable)

Example:

```
>>> a = ( i * 2 for i in range(2 ** 29) )
>>> a
<generator object <genexpr> at 0x7fa9195c5d80>
>>> sum(a)
72057593769492480
```

Note: there is **no** tuple comprehension

Documentation:

- > list comprehensions
- > dict comprehensions
- > generator expressions

