NPRG065: Programming in Python *Lecture 2*

http://d3s.mff.cuni.cz

Department of Distributed and Dependable Systems

D3S

Tomas Bures

Petr Hnetynka

{bures,hnetynka}@d3s.mff.cuni.cz

CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

Comments

```
# this is a comment till end of line

'''
this is a multi line
comment
'''
```

This is not exactly a comment
See later

Documenting code

- a string literal that occurs as the first statement in a function (module, class,…)

```
def sum(a, b):
    """
    Sums two numbers.

    :param a: First number to sum
    :param b: Second number to sum
    :return: Sum of the parameters
    """
    return a + b
```

- many tool for documentation generation
  - pydoc
  - Sphinx
  - …

Department of Distributed and Dependable Systems

Numbers and operators

- int 1, 2, 3,…
- float 1.2, 5.0,…

- common set of operators
  - + - * / %
  - common precedence, can be changed via parentheses
- “uncommon” operators
  - // floor division (common division always returns float)
  - ** power
- int – “unlimited” size

Not exactly true
Will be later

Numbers

- bool
  - subclass of int
  - (almost) anything can be used as bool value
    - more details later
  - bool literals: True, False
- other numeric types
  - complex, Decimal, Fraction
    - more details later

Department of Distributed and Dependable Systems

Blocks

- No begin/end or {}
- **Indentation**

```
while i < 10:
    while j < 10:
        print(i, j)
        i = i + 1
        j = j + 1
```

- The same indentation ⇨ the same block
  - no prescribed amount of spaces (4 are common)
- Single statement per line
  - semicolon can be used but no one uses them
    - only in “one-liners”

```
python -c "import sys; print(sys.version)"
```

Single line

- Single statement per line
  - lines can be “extended” by \<new_line>

```
1 + 2 \
+ 3
```

  - But single line comments (#) cannot be extended this way
  - parentheses can be also used for breaking expressions

```
( 1 + 2 +
3 )
```

Basic control structures

- Like other languages
- if, else, elif

```
if i < 10:
    print('Too small')
else:
    print('OK')
```

```
if i <= 0:
    print('Too small')
elif i > 0 and i < 10:
    print('OK')
else:
    print('Too big')
```

- while

```
while i < 10:
    while j < 10:
        print(i, j)
        i = i + 1
        j = j + 1
```

- Ternary operator

```
a = 0 if i < 10 else 1
```

Distributed and Dependable Systems

Basic control structures

- for

```
for item in something_iterable:
    body
```

  - common usage of for

```
for i in range(10):
    print(i)
```

    - range(n) returns “something like an array” with values from 0 till n-1
      - range(m, n) – values from m till n-1
      - range(m, n, k) – values from m till n-1 with step k

```
range(5, 10)
  5, 6, 7, 8, 9

range(0, 10, 3)
  0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

Loops – break, continue, else

- **`break`**, **`continue`**
  - like C, Java,…
- a loop’s **`else`** clause runs when no break occurs

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:
        print(n, 'is a prime number')
```

```
for num in range(2, 10):
    if num % 2 == 0:
        print("Found an even number", num)
        continue
    print("Found a number", num)
```

`pass` statement

- does nothing
  - sometimes required syntactically

```
while True:
    pass
```

Strings

- `str`
  - immutable sequences of Unicode code points
- String literals
  - single quotes
    - `'allows embedded "double" quotes'`
  - double quotes
    - `"allows embedded 'single' quotes".`
  - triple quoted – may span multiple lines (including new lines)
    - `'''Three single quotes'''`
    - `"""Three double quotes"""`

No new line here

```
print("""\
Usage: my_program [OPTIONS]
     -h                       Display this usage message
     -H hostname              Hostname to connect to
""")
```

Strings

- \ escaping
  - `print('First line.\nSecond line.')`
  - `print('"Isn\'t," they said.')`
  - `print("\"Isn't,\" they said.")`
- raw strings
  - with r prefix
  - no interpretation of “special characters”
  - `'C:\some\name'`
  - `r'C:\some\name'`

New line here

No new line

Strings

- broken strings ~ joining
  - `("spam " "eggs") == "spam eggs"`
    - whitespaces only in between
  - works only with literals

```
s = 'Py'
s 'thon'
```

Error here

- operators with strings
  - + concatenation
  - * repeating

```
s = 'nut'
print(2 * 'co' + s)
```

Work with variables and literals

Strings

- accessing characters

```
word = 'Python'
word[0]  # -> 'P'
word[5]  # -> 'n'
```

- negative numbers – indexing from the right

```
word[-1]  # -> 'n'
word[-2]  # -> 'o'
word[-6]  # -> 'P'
```

Indexing visualized

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

- slicing

```
word[0:2]  # -> 'Py'
word[2:5]  # -> 'tho`
word[:2]   # -> `Py'
word[2:]   # -> 'thon'
word[-2:]  # -> 'on'
```

Strings

- indexing over the bounds

```
word[42]    # -> ERROR
word[4:42]  # -> 'on'
word[42:]   # -> ''
```

- length

```
len(word) # -> 6
```

The builtin function len() is applicable to anything that semantically has a length

Strings

- many functions
  - see documentation

```
'    spaces    '.strip()  #  ->  'spaces'
'Hello world'.split()     #  ->  ['Hello', 'world']
'Python'.find('th')       #  ->  2
'Python'.endswith('on')   #  ->  True
...
```

- testing substrings

```
'Py' in 'Python'          # -> True
```

Department of Distributed and Dependable Systems

Strings

- Formatting strings

{<number>} is replaced with corresponding positional argument

```
print('{0} + {1} = {2}'.format(1, 2, 1 + 2))
                                # -> 1 + 2 = 3
```

```
print('{} + {} = {}'.format(1, 2, 1 + 2))
                                # -> 1 + 2 = 3
```

If arguments used in sequence, numbers can be skipped

Similar formatting characters like in C's printf

```
print('int: {0:d};  hex: {0:x}'.format(42))
                          # -> int: 42;  hex: 2a
print('float: {0:.2f}'.format(1/3))
                          # -> float: 0.33
```

Strings

- formatted strings
  - since Python 3.6
  - prefixed by f

Existing objects

```
var = 42
print(f'int: {var:d};  hex: {var:x} ')

number = 1024
print(f'{number:#0x}')    # -> 0x400
```

See
strings.py

Formatting mini-language

- Details at
  - https://docs.python.org/3.8/library/string.html#formatspec

- Grammar
  - format_spec ::= [[fill]align][sign][#][0][width][grouping_option][.precision][type]
  - fill ::= <any character>
  - align ::= "<" | ">" | "=" | "^"
  - sign ::= "+" | "-" | " "
  - width ::= digit+
  - grouping_option ::= "_" | ","
  - precision ::= digit+
  - type ::= "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"

Strings

- "Historical" note – strings in **Python 2**
  - two types
    - **`str`** – ASCII
      - `'string literal'`
    - **`unicode`** – Unicode
      - `u'unicode literal'`
        - `u` prefix in Python3 can be used with no meaning (backward compatibility)

- In Python 3, all strings are Unicode
- ASCII strings are called byte strings
  - prefixed with `b`

```
b'I am a string'
```

Department of Distributed and Dependable Systems D3S