プログラミング言語 3
オブジェクト指向プログラミング (Python)
Programming Languages 3
Object-Oriented Programming (Python)

田浦

目次

1. 目的 / Objectives
2. Python 最初の概要 / Getting Started with Python
3. Python いくつかの特徴 / Some features of Python
4. クラスによる新しいデータの定義 / Defining new data types with classes
5. オブジェクト指向的な考え方 / Object-Orientated paradigms

Contents

1. 目的 / Objectives
2. Python 最初の概要 / Getting Started with Python
3. Python いくつかの特徴 / Some features of Python
4. クラスによる新しいデータの定義 / Defining new data types with classes
5. オブジェクト指向的な考え方 / Object-Orientated paradigms

目的

- Python を用いてオブジェクト指向の考え方を学ぶ
- まずはオブジェクト指向云々を抜きにして，Python の基本を身につけるための演習を行う
- 本題と直接関係ないが, Python は,
  - (事実) アメリカの大学で初学者に最も教えられている
  - 色々なものを飲み込む言語 (有用なライブラリは必ず “Python interface” があると期待していいかも)
  - (私見) 単純で仕様が「理にかなってる」(頭に来ることが少ない)

Objectives

- learn object-oriented programming via Python
- start with Python without object-oriented features
- off topic: Python
  - is the most taught language in US universities
  - "swallows" everything (want to use this library? $\rightarrow$ it is likely to have its "Python interface")
  - is simple and "reasonable" in most aspects (the spec never irritates me)

Contents

1. 目的 / Objectives
2. Python 最初の概要 / Getting Started with Python
3. Python いくつかの特徴 / Some features of Python
4. クラスによる新しいデータの定義 / Defining new data types with classes
5. オブジェクト指向的な考え方 / Object-Orientated paradigms

Python 文法・データ型の骨子

- 豊富な組込みのデータ型
  - None, 数, 文字列, タプル, リスト, 辞書 (連想記憶), 集合
- 様々なデータ型を簡潔に表す式
- 動的な型付け
- オブジェクト指向
  - 共通の文法 (メソッド名) でデータごとに適した操作
  - クラスを用いて新しいデータ型を定義
  - 動的な型付けと相まった柔軟な再利用
- 「式」(値を持つ) と「文」(値を持たない) が主要な文法記号

Python: basic syntax and data types

- rich builtin data types
  - None, numbers, string, tuple, list, dictionary (associative array), set
- concise expressions to build various data types
- dynamic typing
- object-oriented
  - a common syntax (method) does a different appropriate operation for each data type
  - a class defines a new data type
  - flexible reuse of code, in part due to dynamic typing
- "expression" (that evaluates to a value) and "statement" (that does not result in any value) are distinct syntactic entities

式(1)〜リテラル

- None: (≈ ヌルポインタ)
  - `None`
- 数:
  - `3`, `48.5`, `4.8j`, `100000000000000000000000000000000`
- 文字列:
  - `"Mimura"`, `'He said "Mimura"'`
- タプル:
  - `168,56.5,"Mimura"`
  - `(168,56.5,"Mimura")` (括弧は必須ではないが曖昧さ回避のためもつけるが吉)
- リスト:
  - `[ 3, 4, "Mimura" ]`
- 辞書:
  - `{ "height" : 168, "weight" : 56.5 }`
- 集合:
  - `set([1,2,3])`

Expressions (1) — Literals

- None: (≈ null pointer)
  - `None`
- numbers:
  - `3`, `48.5`, `4.8j`, `1000000000000000000000000000000`
- string:
  - `"Mimura"`, `'He said "Mimura"'`
- tuple:
  - `168,56.5,"Mimura"`
  - `(168,56.5,"Mimura")` (parens are optional but often required to avoid ambiguities)
- list:
  - `[ 3, 4, "Mimura" ]`
- dictionary:
  - `{ "height" : 168, "weight" : 56.5 }`
- set:
  - `set([1,2,3])`

式 (2)

- 変数:
  - `x`
- 演算: (データによって色々な意味になる)
  - `a + b`
- 関数呼び出し:
  - `f(a,b+2,c*2)`, `f(a, b+2, z=c*2)`
- フィールド参照:
  - `a.x`
- メソッド呼び出し:
  - `fp.read(100)`
- 要素, スライス参照:
  - `a[0]`, `d["height"]`, `a[1:2]`
- 無名関数 (ラムダ式):
  - `lambda x:  x + 1`
- リスト内包表記:
  - `[ x * x for x in [1,2,3] ]`
  - `[ x * x for x in range(0,100) if prime(x) ]`

Expressions (2)

- variable:
  - `x`
- operator: (has different meanings depending on types)
  - `a + b`
- function call:
  - `f(a,b+2,c*2)`, `f(a, b+2, z=c*2)`
- field reference:
  - `a.x`
- method call:
  - `fp.read(100)`
- indexing/slicing:
  - `a[0]`, `d["height"]`, `a[1:2]`
- anonymous function (lambda expression):
  - `lambda x:  x + 1`
- list comprehension:
  - `[ x * x for x in [1,2,3] ]`
  - `[ x * x for x in range(0,100) if prime(x) ]`

文(1)〜定義

- (変数への) 代入文

```
1 x = (1,2,3)       # 普通の代入文
2 x,y,z = e         # タプルの要素取り出し
3 [ x,y,z ] = range(0,3) # リストの要素取り出し
```

- 関数定義

```
1 def f(x, y):
2     d = x - y
3     return d * d
```

- 定義のスコープ (有効範囲) については後述

Statements (1) — definitions

- assignment (to variables)

```
1 x = (1,2,3)      # ordinary (single) assignment
2 x,y,z = e        # extract tuple components
3 [ x,y,z ] = range(0,3) # extract list components
```

- function definition

```
1 def f(x, y):
2     d = x - y
3     return d * d
```

- scope rules (where variables are visible/accessible) are described later

文 (2)〜データ構造の更新

- (データ構造への) 代入文

```
1 a = [ 1, 2, 3 ]
2 d = { "height" : 168, "weight" : 56.5 }
```

```
1 a[1] = 20
2 d["height"] = d["height"] + 2.5
```

- del 文

```
1 del d["height"]
2 del a # 変数a の削除
```

Statements (2) — updating data structures

- assignment (to data structures)

```
1 a = [ 1, 2, 3 ]
2 d = { "height" : 168, "weight" : 56.5 }
```

```
1 a[1] = 20
2 d["height"] = d["height"] + 2.5
```

- delete statement

```
1 del d["height"]
2 del a # delete variable a
```

文 (3)〜単純な文

- return 文

```
1 def f(x):
2   return x + 2
```

- あらゆる式は文でもある

```
1 def f(x):
2   fp.write("x = %s" % x)
```

Statements (3) — simple statements

- return statement

```
1 def f(x):
2   return x + 2
```

- any expression is a statement

```
1 def f(x):
2   fp.write("x = %s" % x)
```

文 (4)～制御構造

- if 文

```
1
2 if x < 0:
3   return -x
4 else:   # 省略可能
5   return x
```

- for 文

```
1 for x in [1,2,3]:
2   print(x)
3   print(x * x)
```

- while 文

```
1 while x > 0:
2   x = x - m
3   q = q + 1
```

- break 文, continue 文
- pass (何もしない文)

Statements (4) — control flows

- if statement

```
1
2  if x < 0:
3    return -x
4  else:   # optional
5    return x
```

- for statement

```
1  for x in [1,2,3]:
2    print(x)
3    print(x * x)
```

- while statement

```
1  while x > 0:
2    x = x - m
3    q = q + 1
```

- `break` statement, `continue` statement
- `pass` (does nothing)

文(5)〜その他

- import 文
  - モジュールを使うための文
- クラス定義
  - 新しいデータ型を定義するための文

それぞれ後述

Statements (5) — misc.

- import statement
  - a statement to use a module
- class definition
  - a statement to define a new data type

will be described below

import 文 (モジュールの使い方)

- ある機能がどのモジュールにあるかを突き止める
  - 標準ライブラリは
    `https://docs.python.jp/3.6/library/index.html`
  - その他, 無数の Python モジュールが追加ダウンロード可能
  - 有名なものの多くはパッケージ化されている. 例:

```
1 $ sudo apt-get python-matplotlib
```

import statement (how to use modules)

- find which module has a function you want
  - standard libraries are in
    `https://docs.python.jp/3.6/library/index.html`
  - numerous modules can be downloaded
  - popular modules have packages (any of them)

```
1 $ sudo apt-get python-matplotlib
2 $ sudo pip3 install matplotlib
3 $ pip3 install --user matplotlib
```

import 文 (モジュールの使い方)

目当てのモジュール及びその中の関数名などがわかったら,

1. 方法 1: 丁寧に「モジュール名 . 名前」で参照

```
1 import heapq
2 heapq.heapify([1,2,3])
```

2. 方法 2: モジュール名を自分で定義

```
1 import heapq as hq
2 hq.heapify([1,2,3])
```

3. 方法 3: 選択的に import

```
1 from heapq import heapify
2 heapify([1,2,3])
```

4. 方法 4: 全て import (乱暴)

```
1 from heapq import *
2 heapify([1,2,3])
```

import statement (how to use modules)

once you found the module you want

1. method 1: use the canonical `module_name.var_name` expression

```
1 import heapq
2 heapq.heapify([1,2,3])
```

2. method 2: import with an alias name

```
1 import heapq as hq
2 hq.heapify([1,2,3])
```

3. method 3: import selected names from a module

```
1 from heapq import heapify
2 heapify([1,2,3])
```

4. method 4: import all names from a module

```
1 from heapq import *
2 heapify([1,2,3])
```

Contents

1. 目的 / Objectives
2. Python 最初の概要 / Getting Started with Python
3. Python いくつかの特徴 / Some features of Python
4. クラスによる新しいデータの定義 / Defining new data types with classes
5. オブジェクト指向的な考え方 / Object-Orientated paradigms

特徴的な点

- 文法の癖 (字下げ)
- ほとんどのエラーは動的 (実行時) に発生
- 文字列リテラルの書きかた
- 文字列に対する値の埋め込み (%演算子)
- コンテナ (タプル, リスト, 辞書, 集合), 列 (文字列, タプル, リスト) に対する共通操作
- リスト内包表記

Pitfalls and points to remember

- indentation is part of syntax
- most errors happen at runtime
- string literals
- embedding values into strings
- common operations on containers (tuple, list, dictionary and set) and sequence (string, tuple, list)
- list comprehension

文法の癖 (字下げ)

- 「複数の文の塊 (ブロック)」を字下げで判定
- 言い換えれば「字下げ量 (空白)」が文法の一部
  - C であれば{ 文 文 ...} と中括弧を使っていたもの
  -

```
1 def f(x, y):
2   for i in range(0,x):
3     print(i)  # for 文の中
4     print(x)  # for 文の中
5   print(y)    # for 文の外, 関数の中
6   print(x+y)  # for 文の外, 関数の中
7 print("hi")   # 関数の外
```

- 字下げを気ままにしてはいけない

```
1 # ダメ
2 def f(x, y, z):
3   print(x)
4    print(y)
```

- 教訓: 字下げは Emacs に任せよ (tab 連打で「合法な字下げの場所」)

Indentation is part of the syntax

- indentation determines statements that make a block
  - in C, braces make a block of statements { S; S; ... }
  -
```
def f(x, y):
  for i in range(0,x):
    print(i)  # within the for
    print(x)  # within the for
  print(y)    # outside the for, within the function def
  print(x+y)  # outside the for, within the function def
print("hi")   # outside the function def
```
  - in other words, indentation (the number of leading spaces) is part of the syntax
- you cannot indent arbitrarily

```
# NG
def f(x, y, z):
  print(x)
   print(y)
```

- use a decent editor and let it do the job

文法の癖 (節目はコロン)

- 要所で, コロン (:) が来る
  - def

```
1 def f(x, y, z):
2   ...
```

  - for

```
1 for x in E:
2   ...
```

  - if

```
1 if x:
2   ...
3 else:
4   ...
```

  - while

```
1 while E:
2   ...
```

- 教訓: Emacs で tab 連打して, 字下げが納得する位置に来なかったら文法エラー, 特にコロンのつけ忘れなどを疑おう

Syntactical notes (colons appear at places)

- a colon appears in many places
  - def

```
1 def f(x, y, z):
2   ...
```

  - if

```
1 if x:
2   ...
3 else:
4   ...
```

  - for

```
1 for x in E:
2   ...
```

  - while

```
1 while E:
2   ...
```

変数のスコープ

- C 言語での復習

```
1  int x = 10;                  /* 大域変数 x */
2  int f() {
3    int x = 20;                /* 局所変数 x（上とは別物）*/
4    printf("x = %d\n", x);     /* 3行目の x を参照 */
5  }
```

- Python も精神は同様. だが, 変数定義の特別な文法がないので, 注意が必要

```
1  x = 10                       # 大域変数 x
2  def f():
3    x = 20                     # あくまで局所変数 x（上とは別物）
4    print("x = %d" % x)        # 3行目の x を参照
```

- C で以下のように書いたら違う意味になることに注意

```
1  int x = 10;      /* 大域変数 x */
2  int f() {
3    /* int */ x = 20;         /* これは変数定義ではなく（1行目のx へ）代入 */
4    printf("x = %d\n", x);    /* 1行目の x を参照 */
5  }
```

Variable scopes

- C language review

```
1  int x = 10;        /* global variable x */
2  int f() {
3    int x = 20;       /* local variable x (different from the above) */
4    printf("x = %d\n", x);  /* refers to x at line 3 */
5  }
```

- Python similarly has global and local variables, but is trickier as it does not have a distinct syntax for variable definition

```
1  x = 10              # global variable x
2  def f():
3    x = 20             # introduce a local variable x (not an assignment to
           the global variable above)
4    print("x = %d" % x)   # refers to x at line 3
```

- note the following C code has a different meaning

```
1  int x = 10;    /* global variable x */
2  int f() {
3    /* int */ x = 20;      /* not a local variable definition, but an
           assignment (to the x at line 1) */
4    printf("x = %d\n", x);  /* refers to x at line 1 */
```

変数のスコープ

- つまりは「変数は関数ごとに閉じている」という, 重要かつ慣れ親しんだ規則が保たれているということ
- 規則: ある関数に変数 x への代入が現れたら, x は局所変数として定義される
- 大域変数への代入をしたければどうするの?

```
1 x = 1
2 def add_to_x(dx):
3   x = x + dx
```

```
1 >>> f(3)
2 UnboundLocalError: local variable 'x' referenced before assignment
```

- 本当にしたいのであれば,

```
1 x = 1
2 def add_to_x(dx):
3   global x #「この関数内のx は大域変数」という宣言
4   x = x + dx
```

Scope of variables

- in short, a familiar rule: "variables are basically local to functions" applies in Python too
- rule: if an assignment to variable x appears within a function, x is a local variable of the function
- what if you want to assign to a global variable?

```
1 x = 1
2 def add_to_x(dx):
3   x = x + dx
```

```
1 >>> f(3)
2 UnboundLocalError: local variable 'x' referenced before assignment
```

- if you are sure you want to do that:

```
1 x = 1
2 def add_to_x(dx):
3   global x # declares ``x within this function is global''
4   x = x + dx
```

ほとんどのエラーは動的 (実行時)

- 実行前 (例えば関数の定義時) に検出されるエラーは, 文法エラー (字下げ, 括弧のとじ忘れ, etc.) くらい
- ほとんどのエラーは実際に実行して判明. 以下はいずれも「定義」自体はエラーにならない
  - 型エラー (`3 + "hello"`)
  - 未定義の変数参照

```
1 def print_name(name):
2   print(nama)
3 def main():
4   print_name("Mimura")
```

  - 存在しないフィールドやメソッドの参照
  - 左辺と右辺で数が合わない代入文

```
1 def add_pair(xy):
2   x,y = xy # ペア (2タプル)をもらう予定
3 def main():
4   add_pair((3,4,5))
```

Most errors are detected dynamically (at runtime)

- errors checked before execution (e.g., when you define a function) are mostly syntax errors
- most errors are detected just when executed. errors below are not detected at definition
  - type errors (e.g., `3 + "hello"`)
  - references to unassigned variables

```
1 def print_name(name):
2   print(nama)
3 def main():
4   print_name("Mimura")
```

  - reference to non-existing fields or methods
  - tuple assignments with a wrong number of elements

```
1 def add_pair(xy):
2   x,y = xy # meant to receive a pair
3 def main():
4   add_pair((3,4,5))
```

文字列リテラル

- 文字列リテラルのクオートは一重 (’), 二重 (") のどちらも可
- ’ または"を 3 つ続けると複数行にわたる文字列リテラルも可能

```
"""3重クオートで複数行
にわたるのもOK
"""
```

String literals

- Python allows both a single quote (') or a double quote (") to form a string literal
- triple quotes (''' or """) can make a string literal having multiple lines

```
1 """triple quotes can make
2 muli-line strings
3 """
```

文字列への値の埋め込み

- 「文字列 % 値」で，文字列中の placeholder への値の埋め込み
- 「文字列 % タプル」

```
>>> "name = %s, weight = %.3f" % ("mimura", 100.0/3.0)
'name = mimura, weight = 33.333'
```

- 「文字列 % 辞書」

```
>>> "name = (%(name)s, weight = %(weight).3f"
                % { "name" : "mimura", "weight" : 100.0/3.0 })
'mimura, weight = 33.333'
```

- 「文字列 % それ以外」

```
>>> "age = %d" % 30
'age=30'
```

Embedding values into a string

- *string* `%` *expr* will replace the placeholder in *string* with the value of *expr*

```
>>> "x = %.3f" % (1.2 + 3.4)
'x = 4.6'
```

- if the value of *expr* is a tuple, it replaces multiple placeholders

```
>>> "name = %s, weight = %.3f" % ("mimura", 100.0/3.0)
'name = mimura, weight = 33.333'
```

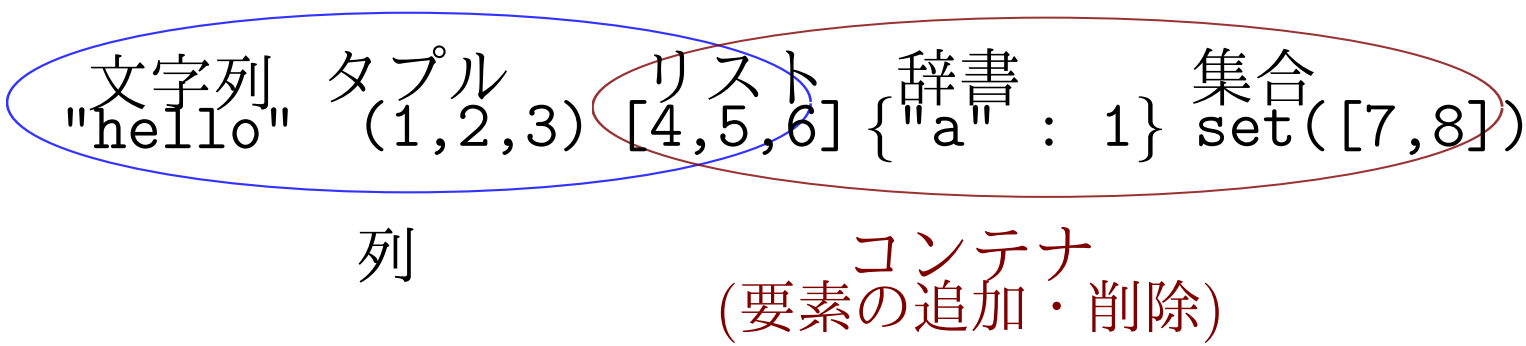
- if the value of *expr* is a dictionary, it replaces named placeholders

```
>>> "name = (%(name)s, weight = %(weight).3f"
                % { "name" : "mimura", "weight" : 100.0/3.0 })
'mimura, weight = 33.333'
```

コンテナおよび列

- 列:
  - 整数の index で要素をアクセスできるデータ構造
  - 組込みの列 = 文字列, タプル, リスト
- コンテナ (入れ物):
  - 複数の要素を保持し, 後から要素の出し入れをできるデータ構造
  - 組込みのコンテナ = リスト, 辞書, 集合

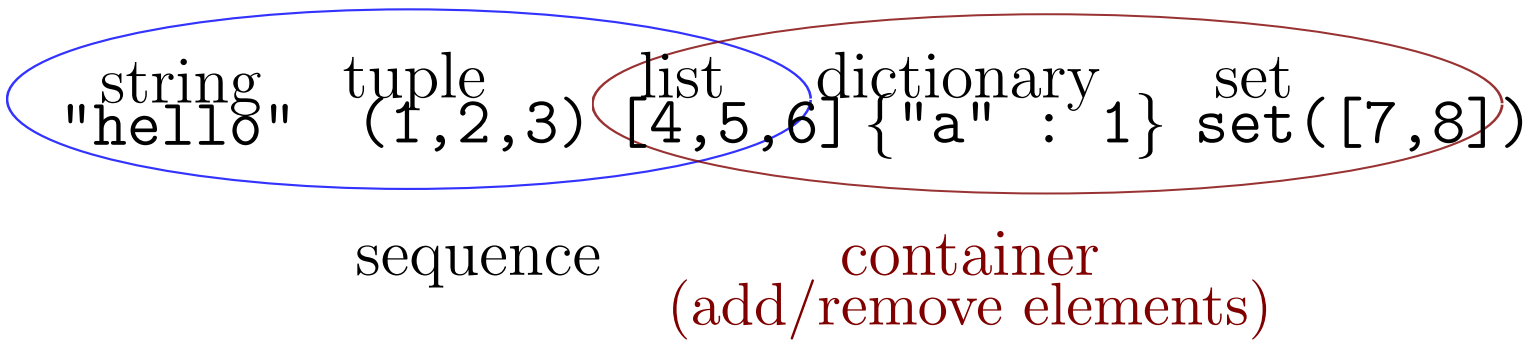
Python は豊富な組込みの列やコンテナを持ち, それらが共通の文法でアクセスできる.



Sequences and containers

- sequences:
  - data structures whose elements can be accessed with integer indices
  - builtin sequences = string, tuple and list
- containers:
  - data structures holding elements and allowing elements to be added/removed
  - builtin containers = list, dictionary and set

Python has a rich set of builtin sequences and containers and a common syntax to access them



列, コンテナに対する式

- 要素参照:
  - `a[3]`, `a[3:5]`, `a[:5]`, `a[3:]`, `a["name"]`
- 要素代入:
  - `a[3] = 5`, `a[3:5] = [1,2]`, `a["height"] = 170.0`
- 要素削除:
  - `del a[3]`, `del a["weight"]`

データが違っても共通の記法で使える．ただし，

- スライス記法 ($[a{:}b]$) は，列のみ可
- 更新 (追加・代入・削除) はコンテナのみ可
- 整数，スライス以外の添字は辞書のみ可

Expressions for sequences and containers

- get item:
  - `a[3]`, `a[3:5]`, `a[:5]`, `a[3:]`, `a["name"]`
- set (add/replace) item:
  - `a[3] = 5`, `a[3:5] = [1,2]`, `a["height"] = 170.0`
- delete item:
  - `del a[3]`, `del a["weight"]`

note that *the syntax is common across different data types*, except for obvious limitations:

- slices ([*a*:*b*]) are valid only for sequences
- updates (addition/assignment/deletion) are valid only for containers
- non-integer indices are valid only for dictionaries

for 文の正体に近づく (1)

- 基本中の基本は, リストの各要素に「文」を実行する

```
1 L = [ 1, 2, 3 ]
2 for x in L:
3     print(x)
```

は

```
1 1
2 4
3 9
```

- 実は列・コンテナなら OK

```
1 for 変数 in 列またはコンテナ:
2     文
3     文
4     ...
```

A closer look at for statement (1)

- the basics: execute the statements for each element of a list

```
1 L = [ 1, 2, 3 ]
2 for x in L:
3     print(x)
```

will produce:

```
1 1
2 4
3 9
```

- actually, it works for any sequence or a container

```
1 for var in a seqeunce or a container:
2     statement
3     statement
4     ...
```

for 文の正体に近づく (2)

- 辞書の場合, 辞書に含まれるキーが取り出される

```
1 D = { "a" :  1, "b" :  2 }
2 for k in D:
3     print(k)
```

の結果は,

```
1 a
2 b
```

- 各要素がタプルなら, それを複数の変数で受けることもできる

```
1 for k,v in [("a",1), ("b",2)]:
2     print("%s = %s" % (k, v))
```

A closer look at for statement (2)

- for a dictionary, a for statement iterates over its *keys*

```
1 D = { "a" :  1, "b" :  2 }
2 for k in D:
3     print(k)
```

will produce:

```
1 a
2 b
```

- if each element is a tuple, you can assign multiple variables with tuple components

```
1 for k,v in [("a",1), ("b",2)]:
2     print("%s = %s" % (k, v))
```

for 文の頻出イディオム (1)

- items() で辞書のキーと値の組 (タプル) のリストが得られる

```
1 >>> d = { "a" : 1, "b" : 2 }
2 >>> d.items()
3 [('a', 1), ('b', 2)]
```

- これを利用して, 辞書のキーと値を処理

```
1 for k,v in d.items():
2     print("%s is %s" % (k, v))
```

の結果は

```
1 a is 1
2 b is 2
```

Frequent idioms of for statements (1)

- `items()` of a dictionary generates tuples of key and values

```
1 for k,v in d.items():
2     print("%s is %s" % (k, v))
```

will produce

```
1 a is 1
2 b is 2
```

for 文の頻出イディオム (2)

- zip 関数で二つのリストを合わせる

```
1 >>> zip([1,2,3],[4,5,6])
2 [(1, 4), (2, 5), (3, 6)]
```

- これを利用して, 二つのリストの対応要素を一緒に処理

```
1 for x,y in zip([1,2,3],[4,5,6]):
2     print(x * y)
```

の結果は

```
1 4
2 10
3 18
```

Frequent idioms of for statements (2)

- `zip` function fuses two lists into a list of tuples

```
1 >>> zip([1,2,3],[4,5,6])
2 [(1, 4), (2, 5), (3, 6)]
```

- useful to process corresponding elements of two lists together

```
1 for x,y in zip([1,2,3],[4,5,6]):
2     print(x * y)
```

will produce:

```
1 4
2 10
3 18
```

リスト内包表記

- for「文」の「式」バージョンと思えば良い
- 例 1

```
>>> [ x * x for x in [ 1, 2, 3 ] ]
[ 1, 4, 9 ]
```

- 例 2 (フィルタ付き)

```
>>> [ x * x for x in range(0,10) if x % 2 == 0 ]
[ 0, 4, 16, 36, 64 ]
```

- ほぼ読んで字のごとく,

```
[ 式 for 変数 in リスト ]
```

は「リスト」の各要素に対し「変数」をその値とした上で「式」を評価したリストを作る

```
[ 式 for 変数 in リスト if 条件式 ]
```

は「条件式」を満たしたものだけを残す

List comprehension

- the "expression" version of for statements
- Ex 1

```
>>> [ x * x for x in [ 1, 2, 3 ] ]
[ 1, 4, 9 ]
```

- Ex 2 (with filters)

```
>>> [ x * x for x in range(0,10) if x % 2 == 0 ]
[ 0, 4, 16, 36, 64 ]
```

- the syntax speaks for itself

```
[ expr for x in lexpr ]
```

makes a list of *"expr"*, with each value in the *lexpr* as x

```
[ expr for x in lexpr if cexpr ]
```

will leave only elements satisfying the *cexpr*.

リスト内包表記

- 想像通り, “for” 以降は for 文が持っていたのと同じ一般性を持つ
- リストは例えば辞書でも良い

```
>>> [ k for k in { "a" : 1, "b" : 2 } ]
[ 'a', 'b' ]
```

- 複数の変数を並べることもできる

```
>>> [ x + y for x,y in zip([1,2,3],[4,5,6]) ]
[ 5, 7, 9 ]
```

- Python も OCaml 同様「少ない学習」で使い始められる言語だが, どうせなら便利なものは覚えたほうが良い
- リスト内包表記は便利なものの代表

List comprehension

- as you will imagine, a list comprehension has a similar generality as a for statement
- *lexpr* part can be a dictionary, for example

```
1 >>> [ k for k in { "a" : 1, "b" : 2 } ]
2 [ 'a', 'b' ]
```

- it can take multiple variables

```
1 >>> [ x + y for x,y in zip([1,2,3],[4,5,6]) ]
2 [ 5, 7, 9 ]
```

- You can live without them, but they are worth mastering

Contents

1. 目的 / Objectives
2. Python 最初の概要 / Getting Started with Python
3. Python いくつかの特徴 / Some features of Python
4. クラスによる新しいデータの定義 / Defining new data types with classes
5. オブジェクト指向的な考え方 / Object-Orientated paradigms

クラス：新しいデータの定義

Python クラス定義の例:

```
1 class point:
2     def __init__(self):
3         self.x = 0
4         self.y = 0
5     def move(self, dx, dy):
6         self.x += dx
7         self.y += dy
```

その利用例

```
1 p = point()
2 p.move(1,2)
3 p.move(3,4)
4 print(p.x,p.y)
```

実行結果

```
1 4 6
```

Defining a new type with class

An example Python class definition:

```
1 class point:
2     def __init__(self):
3         self.x = 0
4         self.y = 0
5     def move(self, dx, dy):
6         self.x += dx
7         self.y += dy
```

with its use:

```
1 p = point()
2 p.move(1,2)
3 p.move(3,4)
4 print(p.x,p.y)
```

and the results:

```
1 4 6
```

用語

```
1  class point:
2      def __init__(self):
3          self.x = 0
4          self.y = 0
5      def move(self, dx, dy):
6          self.x += dx
7          self.y += dy
```

```
1  p = point()
2  p.move(1,2)
3  p.move(3,4)
4  print(p.x,p.y)
```

- クラス
  - データの「形」を定義する
  - 類似物: C の struct, 他の言語の record
- オブジェクト，インスタンス
  - クラス定義に基づいて生まれたデータ
  - 類似物: C の struct 型の変数, malloc された領域
- 一つのクラスからいくらでもオブジェクトが生まれうる

Terminologies (classes and objects/instances)

```
1  class point:
2      def __init__(self):
3          self.x = 0
4          self.y = 0
5      def move(self, dx, dy):
6          self.x += dx
7          self.y += dy
```

```
1  p = point()
2  p.move(1,2)
3  p.move(3,4)
4  print(p.x,p.y)
```

- class
  - defines a "format" of data
  - analogous to: struct in C, record in other languages
- objects or instances
  - data created according to a class definition
  - analogous to: variables of a struct in C, regions allocated by malloc
- a class can have many objects that belong to it

用語

```
1 class point:
2     def __init__(self):
3         self.x = 0
4         self.y = 0
5     def move(self, dx, dy):
6         self.x += dx
7         self.y += dy
```

```
1 p = point()
2 p.move(1,2)
3 p.move(3,4)
4 print(p.x,p.y)
```

- メソッド
  - クラスのオブジェクトに結び付けられた関数
  - Python のメソッド定義の文法は普通の関数定義と同じ．第一引数 (例での `self`) にオブジェクトが渡される
  - メソッド呼び出しの文法

```
1 <<式>>.メソッド名(...)
```

  - <<式>>の結果は「メソッド名」を持つオブジェクト
  - そのオブジェクトが呼び出されたメソッドの第一引数 (例での `self`) になる

Terminologies (methods)

- methods ≈ functions attached to objects of a class

```
1 class point:
2     def __init__(self):
3         self.x = 0
4         self.y = 0
5     def move(self, dx, dy):
6         self.x += dx
7         self.y += dy
```

```
1 p = point()
2 p.move(1,2)
3 p.move(3,4)
4 print(p.x,p.y)
```

- a method definition has the same syntax as an ordinary function definition; the first parameter (`self` in the example) receives the object
- the syntax to call a method

```
1 expr.method_name(...)
```

- the result of *expr* should be an object that has a method named *method_name*
- the object becomes the first parameter of the called method (`self` in the example)

用語

```
1 class point:
2     def __init__(self):
3         self.x = 0
4         self.y = 0
5     def move(self, dx, dy):
6         self.x += dx
7         self.y += dy
```

```
1 p = point()
2 p.move(1,2)
3 p.move(3,4)
4 print(p.x,p.y)
```

- コンストラクタ
  - クラスのオブジェクトを作る関数
  - クラス名と同じ名前の関数
  - デフォルトでは 0 引数の関数
  - `__init__`という名前のメソッド定義をすると，コンストラクタを自由に定義 (カスタマイズ) できる
- 属性，フィールド
  - C の struct のフィールドと同じ
  - 文法:

```
1 <<式>>.フィールド名
```

Terminologies (constructors, attributes/fields)

```
1  class point:
2      def __init__(self):
3          self.x = 0
4          self.y = 0
5      def move(self, dx, dy):
6          self.x += dx
7          self.y += dy
```

```
1  p = point()
2  p.move(1,2)
3  p.move(3,4)
4  print(p.x,p.y)
```

- constructors
  - the function that creates an object of a class
  - in Python, a class has a constructor of the class name
  - by default, it takes no arguments
  - you can define (customize) a constructor by defining a method of the name `__init__`
- attributes or fields
  - analogous to fields of a struct in C
  - syntax to access a field:

```
1  expr.field_name
```

クラス ≈ struct + コンストラクタ + 関数

- 先の定義はCで言うならばだいたい以下と同じ

```
1  typedef struct { int x, y; } point; // data format
2  point * make_point() { // a function to create a point
3    point * p = malloc(sizeof(point));
4    p->x = p->y = 0;
5    return p;
6  }
7  void move(point * p, int dx, int dy) { // a function to work on a point
8    p->x += dx;
9    p->y += dy;
10 }
```

- では何が違う?
  - **多相性:** 同じ名前のメソッド (本例の move) を，クラスごとに異なる中身で定義できる
  - **動的束縛:** 同じ名前のメソッド呼び出しが，オブジェクトが属するクラスにより，適切な (異なる) メソッドを呼び出す
  - **継承:** 既存のクラスを拡張して新しいクラスを定義できる (次のスライド)

a class ≈ a struct + a constructor + functions

- the above class definition is analogous to the following C definitions

```
1  typedef struct { int x, y; } point; // data format
2  point * make_point() { // a function to create a point
3    point * p = malloc(sizeof(point));
4    p->x = p->y = 0;
5    return p;
6  }
7  void move(point * p, int dx, int dy) { // a function to work on a point
8    p->x += dx;
9    p->y += dy;
10 }
```

- so what are the differences?
  - *polymorphism:* different classes can have different definitions of the same method name (`move`)
  - *dynamic binding:* a method call expression will call different methods depending on the class of the object
  - *inheritance:* a class can be extended to define a new class

多相性と動的束縛

```
1  class line:
2      def __init__(self):
3          self.x0 = 0
4          self.y0 = 0
5          self.x1 = 1
6          self.y1 = 1
7      def move(self, dx, dy):
8          self.x0 += dx
9          self.y0 += dy
10         self.x1 += dx
11         self.y1 += dy
```

```
1  o.move(dx, dy)
```

- 同名の (move) メソッドは複数のクラスで定義可能
- メソッド呼び出し (o.move) は, o がどのクラスのインスタンスであるかにより, 適切なメソッドを呼び出す

Polymorphism and dynamic binding

```
1  class line:
2      def __init__(self):
3          self.x0 = 0
4          self.y0 = 0
5          self.x1 = 1
6          self.y1 = 1
7      def move(self, dx, dy):
8          self.x0 += dx
9          self.y0 += dy
10         self.x1 += dx
11         self.y1 += dy
```

```
1  o.move(dx, dy)
```

- multiple classes can define a method with the same name (`move`)
- a method call expression (`o.move`) will call an appropriate one, depending on the class `o` belongs to

クラスの拡張 (継承)

```
1 class circle(point):
2     def __init__(self):
3         # 親クラスのコンストラクタを利用
4         super().__init__()
5         self.r = 1
6     def mag(self, f):
7         self.r = self.r * f
8     def area(self):
9         return self.r * self.r * math.pi
```

```
1 c = circle()
2 c.move(1,2)
3 c.area()
```

- 用語:
  - 親クラス : 継承されるクラス (point)
  - 子クラス : 継承して新しく作られるクラス (circle)
- 子クラスは，親クラスのメソッド定義を自動的に継承
- 子クラスで再定義が可能

Extending a class (inheritance)

```
1 class circle(point):
2     def __init__(self):
3         # call the constructor of the parent class
4         super().__init__()
5         self.r = 1
6     def mag(self, f):
7         self.r = self.r * f
8     def area(self):
9         return self.r * self.r * math.pi
```

```
1 c = circle()
2 c.move(1,2)
3 c.area()
```

- terminologies:
  - parent class : the class to inherit from (`point`)
  - child class : a newly derived class (`circle`)
- the child class inherits methods from the parent
- they can also be redefined in the child class

参考: C++のクラス

```
1  struct point {
2    int x, y;
3    point() { x = 0; y = 0; }
4    void move(int dx, int dy);
5  };
6  void point::move(int dx, int dy) {
7    // ≡ this->x += dx;
8    x += dx;
9    // ≡ this->y += dy
10   y += dy;
11 }
12 int main() {
13   point * p = new point();
14   p->move(3, 4);
15 }
```

- 文法的には，≈ C の struct 内に，メソッドを定義できるようにしたもの
- メソッドは，呼び出されたオブジェクトへのポインタを，this という変数で (暗黙的に) 受け取っている
- メソッドはそのオブジェクト (this) の属性を，通常の変数のようにアクセスできる

Note: C++ classes

```
1  struct point {
2    int x, y;
3    point() { x = 0; y = 0; }
4    void move(int dx, int dy);
5  };
6  void point::move(int dx, int dy) {
7    // ≡ this->x += dx;
8    x += dx;
9    // ≡ this->y += dy
10   y += dy;
11 }
12 int main() {
13   point * p = new point();
14   p->move(3, 4);
15 }
```

- $\approx$ struct of C + method definitions within it
- a method receives the (pointer to) the called object, implicitly in `this` parameter
- a method can access attributes of the object like ordinary variables (without `this`)

Contents

1. 目的 / Objectives
2. Python 最初の概要 / Getting Started with Python
3. Python いくつかの特徴 / Some features of Python
4. クラスによる新しいデータの定義 / Defining new data types with classes
5. オブジェクト指向的な考え方 / Object-Orientated paradigms

オブジェクト指向的な考え方

1. **モジュール化・抽象化:**
   - 「データ型 + 外から呼ばれる手続き (インタフェース)」をワンセットの部品としてソフトを構築していく ⇒ クラス
   - 「外から呼ばれる手続き」の意味にだけ依存したコードは，部品の中身 (実装) が変わっても動きつづける
2. **多相性の利用:**
   - クラスが異なれば同じ名前で違うメソッドを定義できる ⇒ 「同じ使い方の部品」をたくさん作れる
   - どのメソッドが呼ばれるかは，呼ばれているオブジェクトのクラスで決まる
3. **部品の再利用・拡張性:**
   - 既存の実装を拡張して，新しい実装を作ることができる
   - 「使い方が同じ」であれば昔の部品しか知らないコードも動き続ける

Object-Oriented paradigms

1. *modularization and abstraction:*
   - build a software with "data type + externally visible methods (interface)" as a unit ⇒ class
   - code continues to work even if a component changes, as long as it depends only on externally visible methods
2. *polymorphism:*
   - different classes can differently define methods of the same name ⇒ there can be many components of the same interface
   - class of the called object determines which method gets called
3. *reusable and extendable components:*
   - new implementation of an interface can be derived by extending an existing class
   - code that only knows about existing components still works as long as the new component has the same interface (and semantics)

Python に「組み込まれた」オブジェクト指向的考え方

- リスト，タプル，辞書，文字列などで，要素へアクセスする記法が共通
  - `a[idx]`, `a[idx] = x`, `del a[idx]`, ...
- 同じ記法 (例: +) が色々なデータ型に適用でき, 似てはいるが異なる実装を持つ
  - 数値 + 数値 ⇒ 足し算
  - リスト + リスト ⇒ 連結 (タプル，文字列も同様)
  - リスト * 数値 ⇒ リストの繰り返し (タプル，文字列も同様)
- イテレータであれば何でも受け付ける for 文
- 実はそれら (`[]`, `del`, `+`, `for`, ...) もメソッド呼び出しであり, 適切なメソッドを持つクラスを定義すれば，同じ表記で利用可能

Object-Orientation built into Python

- a common syntax to access various data structures (list, tuple, dictionary, string, etc.)
  - `a[idx]`, `a[idx] = x`, `del a[idx]`, ...
- the same operator (e.g., +) applies to various data structures but behaves differently (albeit similarly)
  - number `+` number ⇒ arithmetic addition
  - list `+` list ⇒ concatenation (same for tuples and strings)
  - list `*` number ⇒ repeat the list (same for tuples and strings)
- for statement accepts any containers or sequences (more on this later)
- in fact, they (`[]`, `del`, `+`, `for`, ...) all call methods of particular names; if you define a class with methods of these names, the same syntax applies to these classes too

+，*，[]，etc. がデータ型によって違う動作をする

- 「同じメソッドでもクラス毎に異なる実装が可能」というオブジェクト指向的考え方の一部
- 実は，+，*，[] などもメソッドの一種
  - a + b は，a.__add__(b)
  - a[b] は，a.__getitem__(b)
  - $a.x$ すら実は，a.__getattr__("x")
  - など (言語リファレンス「データモデル → 特殊メソッド名」の節を参照)
- 逆に言うと, __add__というメソッドさえ定義すれば, 足し算+が可能

`+`, `*`, `[]`, etc. behave differently depending on data types

- it is an example of object-oriented paradigm: "classes have different implementations of a single method"
- in fact, `+`, `*`, `[]` etc. are all methods
  - a `+` b calls a`.__add__(`b`)`
  - a`[`b`]` calls a`.__getitem__(`b`)`
  - even a`.`x in fact calls a`.__getattr__("`x`")`
  - etc. (see `https://docs.python.org/3.5/reference/datamodel.html` "data model" → "special method names" in the language reference)
- in other words, you define `__add__` method in a class and you can apply the `+` operator to its objects

__add__を定義すれば足し算が可能

- 定義例:

```
1 class vec2:
2   def __init__(s, x, y):
3     s.x = x
4     s.y = y
5   def __add__(s, p):
6     return vec2(s.x + p.x, s.y + p.y)
```

- 使用例:

```
1 >>> p = vec2(1,2) + vec2(3,4)
2 >>> p.x,p.y
3 (4, 6)
```

Define __add__ and you can plus (+) it

- example definition:

```
1  class vec2:
2    def __init__(s, x, y):
3      s.x = x
4      s.y = y
5    def __add__(s, p):
6      return vec2(s.x + p.x, s.y + p.y)
```

- usage definition:

```
1  >>> p = vec2(1,2) + vec2(3,4)
2  >>> p.x,p.y
3  (4, 6)
```

最大限に再利用可能な関数

- 例えば以下の関数:

```
1  def sum(L, v0):
2    v = v0
3    for x in L:
4      v = v + x
5    return v
```

は, L の要素および v0 の間の, 足し算 (+) さえ定義されていれば, どんなリストやコンテナにも適用可能

- 例:

```
1  >>> sum([1,2,3], 0)
2  6
3  >>> sum(["hello", " ", "world"], "")
4  'hello world'
5  >>> p = sum([vec2(1,2), vec2(3,4)], vec(0,0))
6  >>> p.x,p.y
7  (4,6)
```

- これらをかなり気ままに行える理由の一部は, Python が静的な型検査をしない言語だから

Maximally reusable functions

- the following function:

```
1 def sum(L, v0):
2   v = v0
3   for x in L:
4     v = v + x
5   return v
```

  can apply to *any* lists or containers among whose elements (and v0) + operation is defined
- 例:

```
1 >>> sum([1,2,3], 0)
2 6
3 >>> sum(["hello", " ", "world"], "")
4 'hello world'
5 >>> p = sum([vec2(1,2), vec2(3,4)], vec(0,0))
6 >>> p.x,p.y
7 (4,6)
```

- a part of the reason why you can do things like this so readily is Python does not have a static type check

for 文の正体 (1)

- for 文では, 任意の列やコンテナを処理できると述べたが, 本当はさらに一般的
- for 文

```
1 for ... in E:
2   文
3   ...
```

の E は, 以下を満たすものなら何でも良い

- `__iter__()` メソッドを持ち, これがあるオブジェクトを返す
- そのオブジェクト (イテレータ) は, `__next__()` メソッドを持つ
- `__next__()` メソッドは一度呼ばれるごとに処理したい値を順に返す. これ以上処理する値がないときは, `StopIteration` という例外を発生させる

the truth of for statement (1)

- for statement can process arbitrary sequences or containers
- it is in fact more general
- the E of a for statement

```
for ... in E:
  S
  ...
```

can be *any* object satisfying the following.

1. it has `__iter__()` method that returns an object
2. the returned object (iterator) has `__next__()` method
3. `__next__()` method will return an element to process, or raises `StopIteration` exception when there are no more elements to process

for 文の正体 (2)

- つまり for 文:

```
for x in E:
    ...
```

の正体は,

```
it = E.__iter__()
try:
  while 1:
    x = it.__next__()
    ...
except StopIteration:
  pass
```

のこと

the truth of for statement (2)

- that is,

```
for x in E:
    S
```

is equivalent to:

```
it = E.__iter__()
try:
  while 1:
    x = it.__next__()
    S
except StopIteration:
  pass
```

のこと

iteration 可能 (for 文で処理できる) 他のデータ

- ファイル

```
1 fp = open("a.csv")
2 for line in fp:
3   ...
```

- データベースクエリの結果

```
1 import sqlite3
2 co = sqlite3.connect("a.sqlite")
3 for x in co.execute("select x from a"):
4   ...
```

- 色々な「詳細は違えどもともかく繰り返し」を, 同じ for 文で処理でき, そのようなデータを自分で作ることも可能

Other "iterable" data (for statement can process)

- file objects

```
1 fp = open("a.csv")
2 for line in fp:
3   ...
```

- results from a database query

```
1 import sqlite3
2 co = sqlite3.connect("a.sqlite")
3 for x in co.execute("select x from a"):
4   ...
```

- various "iteration over data (with different details)" can be done with a for statement; and you can define new data that can be iterated with a for statement

iteration 可能なオブジェクトを作る

- 例: 3 の倍数と 3 のつく数だけを生成する

```
1 for x in three():
2   ...
```

が, x に 0,3,6,. . . , 31,32,33,. . . ,39 を入れるように

- ステップ 1: 「次の」そのような数を返す next メソッドを持つクラス three_iter を定義
- ステップ 2: クラス three を, __iter__() メソッドが three_iter のインスタンスを返すように定義

```
1 class three:
2   def __iter__(self):
3     return three_iterator()
```

```
1 class three_iterator:
2   def __init__(self):
3     self.x = 0
4   def __next__(self):
5     for x in range(self.x, 41):
6       if x % 3 == 0 or 30 < x < 40:
7         self.x = x + 1
8         return x
9     raise StopIteration
```

Creating iterable objects (canonically)

- Ex: generate all numbers up to 40 that are divided by 3 or have a 3

```
for x in three():
  ...
```

should assign x 0,3,6,. . . , 31,32,33,. . . ,39

- step 1: define a class, `three_iter`, that has `__next__` method that returns the "next" number
- step 2: define a class, `three`, whose `__iter__()` method returns an instance of `three_iter` class

```
class three:
  def __iter__(self):
    return three_iterator()
```

```
class three_iterator:
  def __init__(self):
    self.x = 0
  def __next__(self):
    for x in range(self.x, 41):
      if x % 3 == 0 or 30 < x < 40:
        self.x = x + 1
        return x
    raise StopIteration
```

ジェネレータ：一番手っ取り早いiteration可能なオブジェクト

- 先の例を以下で済ませられる

```
1 def three():
2   for x in range(0, 41):
3     if x % 3 == 0 or 30 < x < 40:
4       yield x
```

```
1 for x in three():
2   ...
```

- ジェネレータ: 関数定義内に, `yield` という文が一度以上現れているもの
- 「`yield` E $\Longleftrightarrow$ 式 E の値を for 文に供給」と理解しておけば良い. 詳しくは:
  - ジェネレータを呼ぶと即座にiterate可能なオブジェクトを返す
  - そのオブジェクトの`__next__`メソッドが呼ばれると, 関数が次の `yield` 文まで実行され, それに渡された値を返す

Generator : a quickest way to create an iterable object

- the following can do all the above example did

```
1 def three():
2   for x in range(0, 41):
3     if x % 3 == 0 or 30 < x < 40:
4       yield x
```

```
1 for x in three():
2   ...
```

- generator: a function that has an `yield` statement in it
- 「`yield` E $\Longleftrightarrow$ supply the value of E to the for statement; to explain:
  - calling a generator immediately returns an iterable object
  - calling the iterable object's `__next__` method will execute the function up to the next `yield` statement and returns the value passed to it

Python まとめ

- オブジェクト指向
  - 多相性: 同じメソッド名でクラスごとに違う実装
  - 一見「組み込み」の操作も実はメソッド呼び出し
    - `+` : `__add__`
    - `[]` : `__getitem__`
    - `for` : `__iter__` と `__next__`
    - etc.
- 動的型検査 (静的型検査をしない)
- 自然に再利用可能なコード (自然なコードが色々なデータに対して動く)

Python summary

- object-oriented
  - polymorphism: different method implementations with the same name
  - apparently “builtin” operations are in fact implemented with method calls
    - `+` : `__add__`
    - `[]` : `__getitem__`
    - `for` : `__iter__` と `__next__`
    - etc.
- dynamically typed (lack of static type checks)
- naturally highly reusable (a code of “a natural look” can work on many data types)