Rozwiązania szkieletowe w tworzeniu aplikacji www

Tomasz Łukaszuk

Katedra Oprogramowania
Wydział Informatyki
Politechnika Białostocka

Temat wykładu:

Język programowania Python

Fundusze Europejskie dla Rozwoju Społecznego
Rzeczpospolita Polska
Dofinansowane przez Unię Europejską
NCBR Narodowe Centrum Badań i Rozwoju

Materiały przygotowane w ramach projektu "PB 5.0 – dostosowanie oferty dydaktycznej Politechniki Białostockiej do potrzeb nowoczesnej gospodarki oraz zielonej i cyfrowej transformacji" (nr umowy FERS.01.05-IP.08-0327/23-00) w ramach programu Fundusze Europejskie dla Rozwoju Społecznego 2021-2027 współfinansowanego ze środków Europejskiego Funduszu Społecznego Plus.

Język programowania Python: Agenda

1. Sprawy organizacyjne
2. Wprowadzenie do Pythona
3. Kodowanie w Pythonie
   - Styl kodowania
   - Natywne typy danych
   - Siła introspekcji
   - Funkcje
   - Klasy
   - Instrukcje sterujące
   - Dekoratory

Sprawy organizacyjne

Sprawy organizacyjne

Wykład:

- Django, Ruby on Rails (teoria i praktyka)
- slajdy po polsku i angielsku
- udostępniane w cez.wi.pb.edu.pl
- zaliczenie na 15 wykładzie

Pracownia specjalistyczna:

- wprawki (5 zajęć) + projekt (9 zajęć)
- projekt w grupach 2-4 osobowych
- projekt w wybranym frameworku
- raporty z postępów prac nad projektem
- ocena na podstawie projektu

Tematy wykładów

1. Podstawy języka Python
2. Podstawy języka Ruby
3. Wprowadzenie do frameworków webowych
4. Konfiguracje i konwencje
5. Modele, mapowanie obiektowo-relacyjne
6. Kontrolery
7. Dyspozytor URL
8. Template'y (szablony)
9. Formularze
10. Moduł administracyjny
11. Dodatkowe moduły
12. Sesje, caching, internacjonalizacja, wdrażanie aplikacji
13. Zaliczenie wykładu

Literatura

Podstawy:

- Django Software Fundation, Django documentation, online: `https://docs.djangoproject.com`
- Ruby on Rails Guides, online: `http://guides.rubyonrails.org`

Uzupełnienie:

- P. Norton, Python: od podstaw, Helion, Gliwice, 2006
- D. Thomas, Agile: programowanie w Rails, Helion, Gliwice, 2008
- D. Flanagan, Y. Matsumoto, Ruby: programowanie, Helion, Gliwice, 2009
- A. Melé, Django: praktyczne tworzenie aplikacji sieciowych, Helion, 2016
- R. S. Pressman, D. Lowe, Web engineering: a practitioner's approach, Boston, McGraw-Hill, 2009

Wprowadzenie do Pythona

Historia

- Opracowany w późnych latach 1980 przez Guido Van Rossum pracującego w Centrum Wiskunde & Informatica w Holandii. Nazwa wywodzi się od 'Monty Python's Flying Circus'.
- 16-10-2000: Python wersja 2.0
- 3-12-2008: Python wersja 3.0. Niekompatybilna wstecz z wersją 2.0, ale wiele cech wersji 3.0 wprowadzono w wersji 2.6
- 01-01-2020: Python 2 został oficjalnie wstrzymany
- 04-02-2025: Python wersja 3.13.2

Filozofia programowania - The Zen of Python I

1. Ładne jest lepsze niż brzydkie.
2. Jawne jest lepsze niż niejawne.
3. Proste jest lepsze niż złożone.
4. Złożone jest lepsze niż skomplikowane.
5. Płaskie jest lepsze niż zagnieżdżone.
6. Rzadkie jest lepsze niż gęste.
7. Liczy się czytelność.
8. Szczególne przypadki nie są na tyle specjalne żeby łamać zasady.
9. Chociaż praktyczność jest ważniejsza od czystości.

Filozofia programowania - The Zen of Python II

10. Błędy nie powinny być przekazywane w milczeniu.
11. O ile jawnie nie zostaną wyciszone.
12. W obliczu dwuznaczności, odrzucić pokusę zgadywania.
13. Powinien być jeden - i najlepiej tylko jeden - oczywisty sposób aby coś zrobić.
14. Chociaż ten sposób może nie być od razu oczywisty, chyba że jesteś Holendrem.
15. Teraz jest lepsze niż nigdy.
16. Chociaż nigdy jest często lepsze niż "right" teraz.
17. Jeśli implementacja jest trudna do wytłumaczenia, jest to zły pomysł.
18. Jeśli implementacja jest łatwa do wyjaśnienia, to może być dobry pomysł.

Dlaczego używać Pythona? I

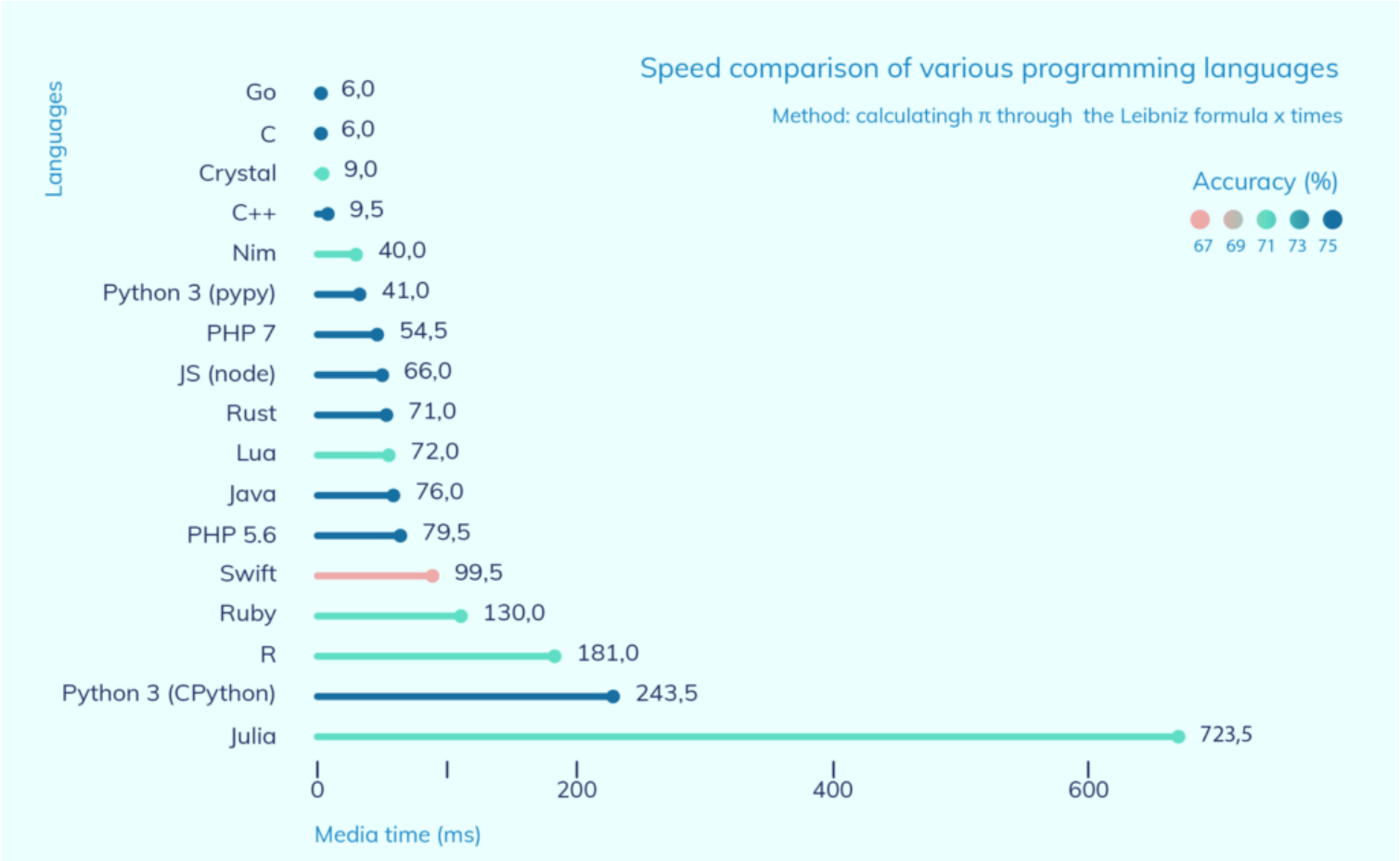
- Zmniejsza czas tworzenia programów
- Bardzo jasna, czytelna składnia
- Bardzo łatwy do nauczenia się
- Bardzo dużo standardowych bibliotek i rozszerzających modułów do wykonywania praktycznie każdego zadania
- Pracuje wszędzie (Windows, Linux/Unix, Mac, Amiga)
- Otwarty i darmowy!

Dlaczego używać Pythona? II

- Dość szybki! (Fractal benchmark, http://www.timestretch.com/FractalBenchmark.html)

Language	exec. time (s)	slowdown
C gcc-4.0.1	0.05	1.00x
Java 1.4.2	0.40	8.00x
Python 2.5.1	9.99	199.80x
Perl 5.8.6 optimized	12.37	247.34x
PHP 5.1.4	23.13	462.40x
Javascript Spider Monkey v1.6	31.06	621.27x
Ruby 1.8.4	34.31	686.18x

Dlaczego używać Pythona? III



Do czego jest używany Python?

- szybkie prototypowanie
- skrypty (w tym skrypty webowe)
- programowanie ad hoc
- aplikacje naukowe
- przetwarzanie XML
- aplikacje bazodanowe
- aplikacje GUI

Kto używa Pythona?

- YouTube
- Uber
- Facebook
- Instagram
- Google
- Pinterest
- PayPal
- Quora
- Disqus
- Netflix
- Spotify
- NASA
- Dropbox

Kodowanie w Pythonie

Cechy języka Python I

- Interpretowalny
- Używa kodu bajtowego (pliki *.pyc i *.pyo)
- WSZYSTKO JEST OBIEKTEM
- Bardzo jasna, czytelna składnia
- Moduły, klasy, funkcje
- Pełna modułowość, wspieranie hierarchii pakietów
- Multi-paradigm: programowanie obiektowe i strukturalne + wiele innych właściwości: programowanie funkcyjne, programowanie aspect-oriented
- Dynamiczne i silne typowanie, polimorfizm, garbage collector, późne wiązania

Cechy języka Python II

- Duck typing: ’when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.’ - James Whitcomb Riley
- Przeciążanie operatorów
- Wcięcia do określenia struktury bloków kodu
- Silne zdolności introspekcji
- Rozszerzenia i moduły pisane w C, C++ (lub Java z Jython, lub .NET dla IronPython)

Typy danych

- Liczbowe: int, long, float, complex
- String: niezmienialny
- Podstawowe kontenery: listy, słowniki, zbiory (zmienialny), krotki (niezmienialny)
- Inne typy, np. binarne dane, wyrażenia regularne, introspekcja
- Moduły rozszerzające mogą definiować własne typy danych

Pierwszy program w Pythonie

```
1  def buildConnectionString(params):
2      """Build a connection string from a dictionary of parameters.
3      Returns string."""
4      return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
5
6  if __name__ == "__main__":
7      myParams = {"server":"mpilgrim",
8                  "database":"master",
9                  "uid":"sa",
10                 "pwd":"secret",
11                 }
12     print(buildConnectionString(myParams))
```

Output:

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

- This example shows basic uses of functions, strings, tuples, lists and dictionaries
- Note the code indentation!

Wszystko jest obiektem

odbchelper.py

```
1  def buildConnectionString(params):
2      """Build a connection string from a dictionary of parameters.
3      Returns string."""
4      return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

```
1  import odbchelper
2
3  params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
4  print(odbchelper.buildConnectionString(params))
5  # -> server=mpilgrim;uid=sa;database=master;pwd=secret
6
7  print(odbchelper.buildConnectionString.__doc__)
8  # -> Build a connection string from a dictionary of parameters.
9  # -> Returns string.
```

- Note how we access the `__doc__` string - a function is also an object!

Listy (Lists)

```
1  li = ["a", "b", "mpilgrim", "z", "example"]
2  li
3  # -> ['a', 'b', 'mpilgrim', 'z', 'example']
4
5  li[4]
6  # -> 'example'
7
8  li[-1]
9  # -> 'example'
10
11 li[1:3]
12 # -> ['b', 'mpilgrim']
13
14 li.append("new")
15 li
16 # -> ['a', 'b', 'mpilgrim', 'z', 'example', 'new']
17
18 li = ['a', 'b', 'mpilgrim']
19 li = li + ['example', 'new']
20 li
21 # -> ['a', 'b', 'mpilgrim', 'example', 'new']
```

- Elementy listy mogą być dowolnego typu
- Listy SĄ UPORZĄDKOWANE

Wyrażenia listowe

```
1  def buildConnectionString(params):
2      """Build a connection string from a dictionary of parameters.
3      Returns string."""
4      return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

- Wygodny sposób na tworzenie list
- Wyrażenie listowe składa się z wyrażenia, następującej po nim klauzuli `for`, a potem zero lub więcej klauzul `for` lub `if`.

```
1  vec1 = [2, 4, 6]
2  vec2 = [4, 3, -9]
3
4  [(x, x**2) for x in vec1]
5  # -> [(2, 4), (4, 16), (6, 36)]
6
7  [x*y for x in vec1 for y in vec2]
8  # -> [8, 6, -18, 16, 12, -36, 24, 18, -54]
9
10 [x+y for x in vec1 for y in vec2]
11 # -> [6, 5, -7, 8, 7, -5, 10, 9, -3]
12
13 [vec1[i]*vec2[i] for i in range(len(vec1))]
14 # -> [8, 12, -54]
```

Słowniki (Dictionaries)

```
1  d = {"server":"mpilgrim", "database":"master"}
2  d
3  # -> {'server': 'mpilgrim', 'database': 'master'}
4
5  d["server"]
6  # -> 'mpilgrim'
7
8  d[24] = 6666
9  d
10 # -> {'server': 'mpilgrim', 'database': 'master', 24 : 6666}
11
12 del d['server']
13 d
14 # -> {'database': 'master', 24 : 6666}
```

- Klucze i wartości słowników mogą być dowolnego typu (zagnieżdżenia)!
- Słowniki NIE SĄ UPORZĄDKOWANE

Krotki (Tuples)

```
1  t = ("a", "b", "mpilgrim", "z", "example")
2  t
3  # -> ('a', 'b', 'mpilgrim', 'z', 'example')
4
5  t[0]
6  # -> 'a'
7
8  t[-1]
9  # -> 'example'
10
11 t[1:3]
12 # -> ('b', 'mpilgrim')
13
14 "z" in t
15 # -> True
```

- Elementy krotki mogą być dowolnego typu
- Krotki SĄ UPORZĄDKOWANE
- Krotki nie posiadają metod i są niezmienialne . . .
- . . . ale elementy krotki mogą być!

Zbiory (Sets)

```
1   s1 = set(['one', 'two', 'three'])
2   s2 = set(['two', 'three', 4])
3
4   s1 | s2
5   # -> set([4, 'two', 'three', 'one'])
6
7   s1 ^ s2
8   # -> set([4, 'one'])
9
10  s1 & s2
11  # -> set(['two', 'three'])
12
13  s1 - s2
14  # -> set(['one'])
```

- Obiekt zbiór jest nieuporządkowaną kolekcją niezmienialnych wartości
- Przydatne do testowania przynależności, usuwania duplikatów z sekwencji i obliczania działań matematycznych, takich jak przecięcie, różnica zbiorów i różnica symetryczna.

Operacje wejścia/wyjścia

Pobieranie danych od użytkownika:

```
1  x = input("Podaj liczbe: ")
2  # -> Podaj liczbe: 6
3
4  x
5  # -> '6'
6
7  x = int(x)
8  x
9  # -> 6
```

Wypisywanie wartości na ekranie:

```
1  print("Hello world")
2  # -> Hello world
3
4  print("Hello world " + str(7))
5  # -> Hello world 7
6
7  print("Hello world %d %s" % (7, "xyz"))
8  # -> Hello world 7 xyz
```

Deklarowanie zmiennych

```
1   x
2   # Traceback (innermost last):
3   #   File "<interactive input>", line 1, in ?
4   # NameError: There is no variable named 'x'
5
6   x = 1
7   x
8   # -> 1
9
10  range(7)
11  # -> [0, 1, 2, 3, 4, 5, 6]
12
13  (MO, TUE, WED, THU, FRI, SAT, SUN) = range(7)
14  MO
15  # -> 0
16  TUE
17  # -> 1
18  SUN
19  # -> 6
```

- Nie można deklarować zmiennej bez przypisania jej wartości
- Moża przypisać wiele wartości na raz

Formatowanie napisów (strings) I

```
1  k = "id"
2  v = "XYZ"
3  "%s=%s" % (k, v)
4  # -> id=XYZ
5
6  id = "XYZ"
7  pwd = "secret"
8  print(pwd + " is not a good password for " + id)
9  # -> secret is not a good password for XYZ
10
11 print("%s is not a good password for %s" % (pwd, uid))
12 # -> secret is not a good password for XYZ
13
14 userCount = 6
15 print("Users connected: %d" % (userCount, ))
16 # -> Users connected: 6
17
18 print("Users connected: " + userCount)
19 # Traceback (innermost last):
20 #   File "<interactive input>", line 1, in ?
21 # TypeError: cannot concatenate 'str' and 'int' objects
```

- Silne typowanie: nie można dodawać integer'a do string'a!

Formatowanie napisów (strings) II

```
1  d = { 'pwd' : 'secret', 'id' : 'XYZ' }
2  "%(id)s=%(pwd)s # user id: %(id)s, password: %(pwd)s" % d
3  # -> 'XYZ=secret # user id: XYZ, password: secret'
```

- Można formatować strings używając słowników (nazwanych argumentów)
- Moża użyć każdego argumentu wielokrotnie (albo wcale)

```
1  params = {"server":"zeus", "database":"master", "id":"XYZ", "pwd":"secret"}
2  ["%s=%s" % (k, v) for k, v in params.items()]
3  # -> ['server=zeus', 'id=XYZ', 'database=master', 'pwd=secret']
4
5  ";".join(["%s=%s" % (k, v) for k, v in params.items()])
6  # -> server=zeus;id=XYZ;database=master;pwd=secret
```

- Strings są obiektami - tak jak wszystko inne!

Wszystko jest obiektem

```
1  s = "abc"
2  dir(s)
3  # -> ['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__
       ', '__format__', '__ge__', '__getattribute__', '__getitem__', '
       __getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__', '__le__
       ', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '
       __reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '
       __setattr__', '__sizeof__', '__str__', '__subclasshook__', '
       _formatter_field_name_split', '_formatter_parser', 'capitalize', 'center',
        'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format',
       'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
        'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', '
       rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', '
       splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', '
       upper', 'zfill']
```

- Did I mention that everything is an object. . . ?
- NOTE: Double underscores are only a convention

`type`, `str` i inne pre-definiowane funkcje

```
1  type(1)
2  # -> <type 'int'>
3  li = []
4  type(li)
5  # -> <type 'list'>
6  import odbchelper
7  type(odbchelper)
8  # -> <type 'module'>
9  import types
10 type(odbchelper) == types.ModuleType
11 # -> True
12 str(1)
13 # -> '1'
14 horsemen = ['war', 'pestilence', 'famine']
15 horsemen
16 # -> ['war', 'pestilence', 'famine']
17 str(horsemen)
18 #-> "['war', 'pestilence', 'famine']"
19 str(odbchelper)
20 # -> "<module 'odbchelper' from 'c:\\docbook\\dip\\py\\odbchelper.py'>"
```

- Other introspecting built-ins: `callable`, `isinstance`, `getattr` ...

Opcjonalne i nazwane parametry

```
1  def info(object, spacing=10, collapse=1):
2      #function body...
```

Valid calls:

```
1  info(odbchelper)                      #1
2  info(odbchelper, 12)                  #2
3  info(odbchelper, collapse=0)          #3
4  info(spacing=15, object=odbchelper)   #4
```

1. With only one argument, `spacing` gets its default value of 10 and `collapse` gets its default value of 1.
2. With two arguments, `collapse` gets its default value of 1.
3. Here you are naming the `collapse` argument explicitly and specifying its value. `spacing` still gets its default value of 10.
4. Even required arguments (like `object`, which has no default value) can be named, and named arguments can appear in any order.

*args i **kwargs

```
1  def foo(hello, *args, **kwargs):
2      print(hello)
3      print('arguments:')
4      for each in args:
5          print('arg: %s' % each)
6      print('keyword arguments:')
7      for each in kwargs:
8          print('kwargs: %s=%s' % (each, kwargs[each]))
9
10 if __name__ == '__main__':
11     foo('LOVE', 'one', 'two', kwarg1='three', kwarg2='four')
```

Result:

```
1  LOVE
2  arguments:
3  arg: one
4  arg: two
5  keyword arguments:
6  kwargs: kwarg1=three
7  kwargs: kwarg2=four
```

Klasyczne klasy

```
1  class ClassicClass:
2      pass
3
4  dir(ClassicClass)
5  # -> ['__doc__', '__module__']
```

- A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names (e.g. `__getitem__`). This allows classes to define their own behavior with respect to language operators.
- Note the `__doc__` attribute.
- Classes are also objects!

Klasy w nowym stylu

```
1  class NewStyleClass(object):
2      pass
3
4  dir(NewStyleClass)
5  # -> ['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '
       __getattribute__', '__hash__', '__init__', '__module__', '__new__', '
       __reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '
       __str__', '__subclasshook__', '__weakref__']
```

- Introduced in Python 2.2 to unify types and classes.
- Always inherit directly either from `object` or a built-in type
- In Python 3.x the explicit base class is not required (because everything will subclass `object`).
- Django uses mostly new-style classes.

Klasa vs. instancja klasy

```
1  class A(object):
2      name = 'class A'
3
4  a = A()
5  a.name
6  # -> 'class A'
7  a.instance_name = 'instance a'
8  a.instance_name
9  # -> 'instance a'
10 a.name = 'instance a'
11 A.name
12 # -> 'class A'
13 a.name
14 # -> 'instance a'
```

- Class attributes serve as default values for class instance attributes
- Class instances can have more attributes than their respective classes!

Metody

```
1  class Advanced(object):
2      def __init__(self, name):
3          self.name = name
4
5      @staticmethod
6      def Description():
7          return 'This is an advanced class.'
8
9      @classmethod
10     def ClassDescription(cls):
11         return 'This is advanced class: %s' % repr(cls)
```

- Three types of methods:
  - Instance methods (`__init__`)
  - Static methods (`Description`)
  - Class methods (`ClassDescription`)
- Class methods and static methods require decorators (to be presented later on)

Metody

- Instance methods always receive the instance as the first argument (can acces all the attributes of both the instance and the class).
- Class methods always receive the class as the first argument (can access all the attribute of the class only)
- Static methods do not receive either the instance or the class (cannot access either)
- Class methods may be called using either an instance or a class

Wartość logiczna prawda - True

- Każdy obiekt może być testowany na wartość prawdy, do wykorzystania w wyrażeniach `if` lub `while` lub jako operand w operacjach logicznych. Następujące wartości są uważane za fałszywe (wartościowane jako false):
  - `None`
  - `False`
  - zero każdego typu numerycznego, np. `0, 0L, 0.0, 0j`
  - każda pusta sekwencja, np. `'', (), []`
  - każde puste mapowanie, np. `{}`
  - instancje klas zdefiniowanych przez użytkownika, jeśli klasa definiuje metodę `__nonzero__()` lub `__len__()`, gdy ta metoda zwraca wartość całkowita zero lub boolowską False.
- Wszystkie inne wartości są uważane za prawdziwe

Operacje boolowskie

Operation	Result
x `or` y	if *x* is false, then *y*, else *x*
x `and` y	if *x* is false, then *x*, else *y*
`not` x	if *x* is false, then `True`, else `False`

- `or` wartościuje drugi argument tylko w przypadku kiedy pierwszy argument jest False
- `and` wartościuje drugi argument tylko w przypadku kiedy pierwszy argument jest `True`
- Należy pamiętać, że operatory `or`/`and` mogą zwracać nie tylko `True` lub `False`

Konstrukcja `if`

```
1  x = int(input("Please enter an integer: "))
2  # -> Please enter an integer: 42
3  if x < 0:
4      x = 0
5      print('Negative changed to zero')
6  elif x == 0:
7      print('Zero')
8  elif x == 1:
9      print('Single')
10 else:
11     print('More')
12
13 # -> More
```

- Może być zero lub więcej części `elif`
- Część `else` jest opcjonalna

Konstrukcja `while`

```
1  a, b = 0, 1
2  while b < 1000:
3      print(b, end=' ')
4      a, b = b, a+b
5
6  # -> 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

- Nic nowego . . .
- end=’ ’ zapobiega przejściu do nowej linii.

Konstrukcja `for`

```
1  # Measure some strings:
2  a = ['cat', 'window', 'defenestrate']
3  for x in a:
4      print(x, len(x))
5
6  # -> cat 3
7  # -> window 6
8  # -> defenestrate 12
```

- Zamiast zawsze iterować po rosnących liczbach arytmetycznych (jak w Pascalu), lub dając użytkownikowi możliwość definiowania zarówno kroku iteracji jak i momentu jej zatrzymania (jak w C), w Pythonie iteracja odbywa się po ciągu elementów dowolnej kolejności (lista lub string), w kolejności, w jakiej występują w sekwencji.
- Nie jest bezpieczne modyfikowanie sekwencji po której następuje iteracja.

Wyjątki i obsługa wyjątków

```
1  for arg in sys.argv[1:]:
2      try:
3          f = open(arg, 'r')
4      except IOError:
5          print('cannot open', arg)
6      else:
7          print(arg, 'has', len(f.readlines()), 'lines')
8          f.close()
9      finally:
10         print("Finished with the file")
```

- A `try` statement may have more than one except clause
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped, the respective `except` clause is executed, and then execution continues after the `try` statement.
- The `try ... except` statement has an optional `else` clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception.
- The `finally` clause is executed no matter what happens (even when an unhandled exception is raised)

Dekoratory: idea

A decorator is usually a function that transforms another function:

```
1  def decorator_function(target):
2      # Do something with the target function
3      target.attribute = 1
4      return target
5
6  def target(a,b):
7      return a + b
8
9  #This is how we apply the decorator
10 target = decorator_function(target)
```

In this simple case, the decorator just adds an attribute to the function being decorated:

```
1  target(1,2)
2  # -> 3
3
4  target.attribute
5  # -> 1
```

Do czego używać dekoratorów?

- We need to change functions and methods: to add synchronisation, to add logging, etc
- This was possible before, but changes had to be made in places other than the declaration and could be hard to find later on
- It is reasonable to group them with the declaration of a function
- Since Python 2.2 two decorators were added: classmethod and staticmethod
- Adding syntax has been difficult, as this should be rather simple, not scarying for newcommers, give clear intent, and be clearly visible
- Java style decorators (decorator) were added in Python 2.4a2

Java-like syntax

Since Python 2.4a2, the code shown earlier can be rewritten as follows:

```
1  def decorator_function(target):
2      # Do something with the target function
3      target.attribute = 1
4      return target
5
6  # Here is the decorator, with the syntax '@function_name'
7  @decorator_function
8  def target(a,b):
9      return a + b
```

- Note that decorator functions are called when they are applied, not when the decorated function is called
- Decrators can do many things (conditional function calling, transforming arguments), but they are not really different from what you've seen above

Funkcje opakowane - Wrapper

```
1  def decorator(target):
2
3      def wrapper():
4          print('Calling function "%s"' % target.__name__)
5          return target()
6
7      # Now we need to assign the atttribute to the *wrapper function*.
8      wrapper.attribute = 1
9      return wrapper
10
11 @decorator
12 def target():
13     print('I am the target function')
```

```
1  target()
2  # -> Calling function "target"
3  # -> I am the target function
4
5  target.attribute
6  # -> 1
```

- The wrapper function can do whatever it wants to the target function

Dekoratory dla funkcji akceptujących argumenty

```
1  def decorator(target):
2
3      def wrapper(*args, **kwargs):
4          kwargs.update({'debug': True}) # Edit the keyword arguments
5          print('Calling "%s" in debug mode' % target.__name__)
6          return target(*args, **kwargs)
7
8      wrapper.attribute = 1
9      return wrapper
10
11 @decorator
12 def target(a, b, debug=False):
13     if debug: print('[Debug] I am the target function')
14     return a+b
```

```
1  target(1,2)
2  # -> Calling "target" in debug mode
3  # -> [Debug] I am the target function
4  # -> 3
5
6  target.attribute
7  # -> 1
```

Dekoratory dla metod instancji

A decorator for a class method should assume that the first argument is always `self`:

```
1   def wrapper(self, *args, **kwargs):
2       # Do something with 'self'
3       print(self)
4       return target(self, *args, **kwargs)
```

Needless to say, an instance method can also be used as a decorator.

Dekoratory akceptujące argumenty

```
1  def options(value):
2      def decorator(target):
3          # Do something with the target function
4          target.attribute = value
5          return target
6      return decorator
7
8  @options('value')
9  def target(a,b):
10     return a + b
```

```
1  target(1,2)
2  # -> 3
3
4  target.attribute
5  # -> 'value'
```

Since the `decorator` function is defined inside the `options` function, it has access to any of the arguments passed to `options`.