Python im Wunderland

the beauty of code
KunterBuntesSeminar WiSe2011

Justus Winter ⟨4winter@informatik.uni-hamburg.de⟩

Universitaet Hamburg - Fachbereich Informatik

2011-11-17

1. python rocks
   - overview
   - philosophy
   - by examples
     - statements and control flow
     - a complete example
   - concepts
     - duck typing
     - list comprehension
     - list operations
     - no information hiding
2. theoretical computer science rocks too
   - Gentzen calculus
     - logic crash course
     - deriving tautologies from thin air
   - fixed-point theorem

python ...

- is a multi-paradigm language (object-oriented, imperative, functional)
- is strongly and dynamic typed, uses duck typing
- was released by Guido van Rossum in 1991
- comes with batteries included
- rocks ;)

Philosophy of perl: Tim Toady

There is more than one way to do it.

Python ...

- is a clear and minimalistic language
- has a consistent and intuitive behavior
- enables the user to write beautiful code
- discourages premature optimizations
- is extensible (c and c++ interface)

Zen of Python

There should be one, and preferably only one, obvious way to do it.

Philosophy of perl: Tim Toady Bicarbonate

There is more than one way to do it, but sometimes consistency is not a bad thing either.

Python ...

- is a clear and minimalistic language
- has a consistent and intuitive behavior
- enables the user to write beautiful code
- discourages premature optimizations
- is extensible (c and c++ interface)

Zen of Python

There should be one, and preferably only one, obvious way to do it.

variables

```
>>> a = 23
>>> type(a)
<type 'int'>
>>> a = 'fourtytwo'
>>> type(a)
<type 'str'>
>>> a + 23
TypeError: cannot concatenate 'str' and 'int' objects
```

if statement

```
>>> if 6 * 9 == 42:
...     print('whoohoo')
... else:
...     print('too bad')
too bad
>>>
```

for statement

```
>>> for i in [0, 1, 2]:
  >     print(i)
0
1
2
>>> for i in range(23):
  >     print(i)
0
1
2
⋮
22
>>>
```

def statement

```
>>> def quackAndWalk(who):
  >     print(who, 'quacks amused and walks away.')
>>> quackAndWalk('Donald')
Donald quacks amused and walks away.
>>> quackAndWalk('Tom')
Tom quacks amused and walks away.
>>> quackAndWalk(23)
23 quacks amused and walks away.
```

def statement (cont)

```
>>> def calculate(x):
...     'Returns 2 * x^2 + x - 3'
...     return 2 * x ** 2 + x - 3
>>> calculate
<function calculate at 0xb7c101ac>
>>> help(calculate)
Help on function calculate in module __main__:

calculate(x)
    Returns 2 * x^2 + x - 3
>>> calculate(0)
-3
>>> calculate(2)
7
```

```python
#!/usr/bin/env python2.7

import logging
import argparse

def main(args):
    logging.debug('About to print a greeting')
    print('Hello %s.' % (args.name))

if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description = 'Prints a nice greeting.')
    parser.add_argument('name',
        help = 'how may I address you?')
    parser.add_argument(
        '-v', '--verbose', action = 'store_true',
        help = 'be verbose')
    args = parser.parse_args()

    logging.basicConfig(level =
        logging.DEBUG if args.verbose else logging.INFO)

    main(args)
```

duck typing

Alex Martelli

If it walks like a duck and quacks like a duck, I would call it a duck.

- duck typing allows polymorphism without inheritance

```
>>> def add_scale(a, b, factor):
  >     return (a + b) * factor
>>> print(add_scale(1, 2, 3))
9
>>> print(add_scale([1, 2, 3], [4, 5, 6], 2)
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
>>> print(add_scale('apples ', 'and oranges ', 2))
apples and oranges apples and oranges
```

list comprehension

mathematical notation

$$a = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$b = \{2 * n | n \in a\}$$

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [n for n in a]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [2 * n for n in a]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> [2 * n for n in a if n % 2 == 1]
[2, 6, 10, 14, 18]
```

set comprehension

mathematical notation

$$a = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$b = \{2 * n | n \in a\}$$

```
>>> a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> {n for n in a}
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> {2 * n for n in a}
set([0, 2, 4, 6, 8, 10, 12, 14, 16, 18])
>>> {2 * n for n in a if n % 2 == 1}
set([2, 6, 10, 14, 18])
```

sequence operations

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[2]
2
>>> a[-2]
8
>>> a[2:5]
[2, 3, 4]
>>> a[23]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> 5 in a
True
```

sequence operations

```
>>> a = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> a[2]
2
>>> a[-2]
8
>>> a[2:5]
(2, 3, 4)
>>> a[23]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
>>> 5 in a
True
```

sequence operations

```
>>> a = 'Hello cruel world.'
>>> a[2]
'l'
>>> a[-2]
'd'
>>> a[2:5]
'llo'
>>> a[23]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> 'o' in a
True
```

no information hiding

Quoting Wikipedia on Information Hiding

In object-oriented programming, information hiding reduces software development risk by shifting the code's dependency on an uncertain implementation (design decision) onto a well-defined interface. Clients of the interface perform operations purely through it so if the implementation changes, the clients do not have to change.

- python has no concept of private variables or functions
- enjoy the *bibo*[1] principle
- . . . after all, we're all consenting adults, right?

[1] bullshit in bullshit out

propositional logic

constants	*true*, *false*
atomic propositions	$A, B, C, on, light\_is\_on$
unary junctor	not $\neg$
binary junctors	and $\wedge$, or $\vee$, material implication $\supset$
delimiters	parenthesis (,)

Examples of formulas:

- $A \wedge B$
- $\neg A \vee B$
- $A \supset B$
- $A \supset (B \supset A)$

interpretations

A	B	$A \wedge B$	$A \vee B$	$\neg A \vee B$	$A \supset B$
false	*false*	*false*	*false*	*true*	*true*
false	*true*	*false*	*true*	*true*	*true*
true	*false*	*false*	*true*	*false*	*false*
true	*true*	*true*	*true*	*true*	*true*

tautologies

Definition: Tautology

A tautology is a formula that is true in all interpretations.
$\models \varphi$ denotes that φ is a tautology.

A	$\neg A$	$A \supset A$ $\neg A \vee A$
false	*true*	*true*
true	*false*	*true*

A	B	$B \supset A$ $\neg B \vee A$	$A \supset (B \supset A)$ $\neg A \vee (\neg B \vee A)$
false	*false*	*true*	*true*
false	*true*	*false*	*true*
true	*false*	*true*	*true*
true	*true*	*true*	*true*

deducing formulas from sets

- $\models \varphi$ is a shortcut for $\varnothing \models \varphi$
- $\Gamma \models \varphi$ means φ can be deduced from a set of formulas Γ
  - $A \wedge (B \supset B)$ is not a tautology, but can be deduced from $\{A\}$:
  - $\{A\} \models A \wedge (B \supset B)$
  - this is equivalent to the formula $A \supset (A \wedge (B \supset B))$ being a tautology
  - in general: $\Gamma \models \varphi \iff \models \bigwedge \Gamma \supset \varphi$
- in fact, all tautologies can be deduced from an empty set: $\varnothing \models \top$
- and anything can be deduced from a contradiction: $\{A, \neg A\} \models \psi$

Gentzens logistischer klassischer Kalkül

In the sequent calculus, a conjunction of assumptions Γ guarantees that a disjunction of propositions Δ holds: $\Gamma \vdash \Delta$.

Axiom	$\dfrac{}{A \vdash A}\ A$	
Weakening	$\dfrac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ W_L$	$\dfrac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\ W_R$
Implication	$\dfrac{\Gamma \vdash X, \Delta \quad \Gamma, Y \vdash \Delta}{\Gamma, X \supset Y \vdash \Delta}\ \supset_L$	$\dfrac{\Gamma, X \vdash Y, \Delta}{\Gamma \vdash X \supset Y, \Delta}\ \supset_R$

Used rules

$$\frac{}{A \vdash A}\ A \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\ W_R \qquad \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ W_L$$

$$\frac{\Gamma \vdash X, \Delta \quad \Gamma, Y \vdash \Delta}{\Gamma, X \supset Y \vdash \Delta}\ \supset_L \qquad \frac{\Gamma, X \vdash Y, \Delta}{\Gamma \vdash X \supset Y, \Delta}\ \supset_R$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\ A}{A \vdash A, B}\ W_R \qquad \dfrac{\dfrac{}{B \vdash B}\ A}{A, B \vdash B}\ W_L}{A, A \supset B \vdash B}\ \supset_L}{A \wedge (A \supset B) \vdash B}\ \wedge_L}{\vdash (A \wedge (A \supset B)) \supset B}\ \supset_R$$

proving that the flying spaghetti monster exists

$$\varphi \wedge \neg\varphi \models \mathit{fsm\_exists}$$

1. every interpretation that is a model for $\varphi \wedge \neg\varphi$ is a model for *fsm_exists*
2. but since there is no model for $\varphi \wedge \neg\varphi$, there is no model from that set that *is not* a model for *fsm_exists*
3. so *fsm_exists* is a semantic consequence of $\varphi \wedge \neg\varphi$

proving that the flying spaghetti monster exists

$$\varphi \wedge \neg\varphi \models \mathit{fsm\_exists}$$

assumption	$\varphi \wedge \neg\varphi$	(1)
conjunction elemination (1)	φ	(2)
conjunction elemination (1)	$\neg\varphi$	(3)
disjunction introduction (2)	$\varphi \vee \mathit{fsm\_exists}$	(4)
disjunctive syllogism (3, 4)	$\mathit{fsm\_exists}$	(5)
thus (1, 5)	$(\varphi \wedge \neg\varphi) \supset \mathit{fsm\_exists}$	(6)

it all began one day . . .

. . . when I - a member of the machine tribe - came across a tale from the λ-tribe. They told me about the *y-combinator*, which led me to wikipedia. And the mighty encyclopædia told me:

```
>>> Z = lambda f: (lambda x: f(lambda *args: x(x)(*args))) \
                  (lambda x: f(lambda *args: x(x)(*args)))
>>> fact = lambda f: lambda x: 1 if x == 0 else x * f(x - 1)
>>> Z(fact)(5)
120
```

o,O

introducing fixed point combinators

A fixed point combinator is a function y that computes a fixed point p of any another given function.

$$p = y(f) \quad \text{so that} \quad f(p) = p$$

or to put it differently

$$f(y(f)) = y(f)$$

One of such functions is Haskell Curry's Y:

$$Y = \lambda f(\lambda x\ f(x\ x))(\lambda x\ f(x\ x))$$

what the fuck is going on here?

Let's have a look at it step by step. We first introduce a function *bottom* that raises an exception when called:

```
>>> bottom()
Traceback (most recent call last):
...
Bottom
>>>
```

what the fuck is going on here?

```
>>> fact = lambda f: lambda x: 1 if x == 0 else x * f(x - 1)
```

fact is a function taking an argument *f* and returning a function.
Note that *f* is later used like a function.
So let's call *fact* with our *bottom* function:

```
>>> fact(bottom)
<function <lambda> at 0x2808cf8>
```

Oh right, *fact* returns a function, so let's call it:

```
>>> fact(bottom)(5)
Traceback (most recent call last): [...] Bottom
```

what the fuck is going on here?

That didn't worked too well, maybe more *fact* helps:

```
>>> fact(bottom)(5)
Traceback (most recent call last): [...] Bottom
>>> fact(fact(bottom))(5)
Traceback (most recent call last): [...] Bottom
>>> fact(fact(fact(bottom)))(5)
Traceback (most recent call last): [...] Bottom
>>> i_many_application(fact, 4)(bottom)(5)
Traceback (most recent call last): [...] Bottom
>>> i_many_application(fact, 5)(bottom)(5)
Traceback (most recent call last): [...] Bottom
```

what the fuck is going on here?

Hm, let's keep trying:

```
>>> i_many_application(fact, 6)(bottom)(5)
120
>>> i_many_application(fact, 7)(bottom)(5)
120
>>> i_many_application(fact, 8)(bottom)(5)
120
```

That's a nice change. Here is *fact* again:

```
>>> fact = lambda f: lambda x: 1 if x == 0 else x * f(x - 1)
```

what the fuck is going on here?

```
>>> fact = lambda f: lambda x: 1 if x == 0 else x * f(x - 1)
```

The sequence of values computed by

$$i\_many\_application(fact,\ n)(bottom)(5)$$

starting at $n = 0$ is

$$[\bot, \bot, \bot, \bot, \bot, \bot, 120, 120, 120, \dots]$$

120 is the lowest upper bound of this sequence and incidentally this is exactly what the fixed point combinator Z defined above computes:

$$Z(fact)(5) = \bigsqcup_{i=0}^{\infty} fact^i(bottom)(5) = 120$$

References

- http://python.org/
- https://en.wikipedia.org/wiki/Python_(programming_language)
- https://en.wikipedia.org/wiki/Duck_typing
- https://en.wikipedia.org/wiki/Information_hiding
- https://en.wikipedia.org/wiki/There%27s_more_than_one_way_to_do_it
- https://en.wikipedia.org/wiki/Principle_of_explosion
- https://en.wikipedia.org/wiki/Gentzen_calculus
- http://homepage.mac.com/farwer/nkl0506/Folien_attachments/V1-NK_LK.pdf
- http://wingolog.org/archives/2011/07/12/static-single-assignment-for-functional-programmers
- https://secure.wikimedia.org/wikipedia/en/wiki/Fixed_point_combinator