

Python

chrysn <chrysn@fsfe.org>

2008-09-25

Introduction

Structure, Language & Syntax

Strengths & Weaknesses

Introduction

Structure, Language & Syntax

Strengths & Weaknesses

Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.

interpreted stack based byte code; JIT available; restricted subset compilable

object-oriented every expression returns an object

high-level “readable pseudo-code”

dynamic semantics ? (suggestions welcome)

Facts & Figures

inventor & BDFL Guido van Rossum

influenced by ABC, ALGOL 68, C, Haskell, Icon, Lisp, Modula-3,
Perl, Java

current version 2.5.2 (2008-02-22)

looking forward to Python 3 in October

currently used by NASA, CERN, Yahoo, Google, YouTube

currently used in Blender, Mailman, Gentoo Linux, scons, Zope

Introduction

Structure, Language & Syntax

Strengths & Weaknesses

Structure

- ▶ Every Python file (*.py) is a Python module.

Structure

- ▶ Every Python file (*.py) is a Python module.
- ▶ Python modules contain statements.

```
1 name = "world"  
2 print "hello"  
3 print name
```


Structure

- ▶ Every Python file (*.py) is a Python module.
- ▶ Python modules contain statements.
- ▶ Statements can contain indented clauses, which contain other statements.

```
1 pi = 3
2 if pi > 3:
3     print " pi > 3"
4 else:
5     print " pi <= 3"
```

Structure

- ▶ Every Python file (*.py) is a Python module.
- ▶ Python modules contain statements.
- ▶ Statements can contain indented clauses, which contain other statements.
- ▶ Statements can contain expressions.

```
1 pi = 3
2 if pi > 3:
3     print " pi > 3"
4 else:
5     print " pi <= 3"
```

Structure

- ▶ Every Python file (*.py) is a Python module.
- ▶ Python modules contain statements.
- ▶ Statements can contain indented clauses, which contain other statements.
- ▶ Statements can contain expressions.
- ▶ An expression is a valid statement.

```
1 window.show()  
2  
3 my_list.append(42)  
4 my_list.sort()
```

Structure

- ▶ Every Python file (*.py) is a Python module.
- ▶ Python modules contain statements.
- ▶ Statements can contain indented clauses, which contain other statements.
- ▶ Statements can contain expressions.
- ▶ An expression is a valid statement.
- ▶ Every expression can be evaluated to an object.

```
1 # function that prints a number squared
2 def f(x):
3     print x ** 2
4
5 print f
6 # <function f at 0x7f1e43f439b0>
7 print f(2)
8 # 4
9 # None
```

Statements: Functions (1/2)

```
1 def move(start, end, velocity=1):
2     return (end-start)/velocity
3
4 t = move(0, 1) # equivalent:
5 t = move(0, 1, 1)
6 t = move(velocity=1, start=0, end=1)
7
8 # anonymous function
9 t = (lambda s, e, v=1: (e-s)/v)(0, 1)
```

- ▶ Arguments can have default values
- ▶ (Default arguments are evaluated when function is defined)
- ▶ Can use positional or named arguments
- ▶ Implicit return None
- ▶ lambda: anonymous functions, one expression only

Statements: Functions (2/2)

```
1 def logging(function):
2     def decorated(*args, **kwargs):
3         print function.__name__, args, kwargs
4         function(*args, **kwargs)
5     return decorated
6
7 @logging
8 def move(start, end, velocity=1):
9     """ Calculate time needed to move """
10    return (end-start)/velocity
```

- ▶ *something is list of positional arguments
- ▶ **something is dictionary of keyword arguments
- ▶ @decorator passes function through another function before it is assigned to its name

Statements: Control Structures

- ▶ `for`: Iteration over iterable object

```
1 for i in range(99, 1, -1):  
2     print "%d bottles of beer..." % i
```

Statements: Control Structures

- ▶ `for`: Iteration over iterable object
- ▶ `while`: Loop while condition is true (no assignments)

```
1 while time.time() < end_time:  
2     # can do another round of calculations  
3     calculateAnotherRound()
```


Statements: Control Structures

- ▶ `for`: Iteration over iterable object
- ▶ `while`: Loop while condition is true (no assignments)
- ▶ `else` clause for loops: executed unless `break` was used.

```
1 for p in PATH:  
2     joint = os.path.join(p, filename)  
3     if os.path.exists(joint):  
4         print "found as %s" % joint  
5         break  
6 else:  
7     raise Exception("Not found in PATH.")
```

Statements: Control Structures

- ▶ `for`: Iteration over iterable object
- ▶ `while`: Loop while condition is true (no assignments)
- ▶ `else` clause for loops: executed unless `break` was used.
- ▶ `if`: Condition; used with `elif` as a replacement for Switch/Case constructs

```
1 if spam_score < 3:  
2     print "Probably ham."  
3 elif spam_score < 6:  
4     print "Might be spam."  
5 else :  
6     print "probably spam."
```

Statements: Control Structures

- ▶ `for`: Iteration over iterable object
- ▶ `while`: Loop while condition is true (no assignments)
- ▶ `else` clause for loops: executed unless `break` was used.
- ▶ `if`: Condition; used with `elif` as a replacement for Switch/Case constructs
- ▶ `try / except / else finally`: Exception handling

```
1 try :  
2     logfile = open(logfilename , 'w')  
3 except IOError:  
4     print "Cannot open log; logging disabled."  
5     logging = False
```

Object Model

- ▶ Every Python object has
 - ▶ an identity (think memory address)
 - ▶ a type (an object)
 - ▶ a value (e. g. property values)
- ▶ Behavior can be defined by metaclass (class's class), class or object
- ▶ There are no methods, only object bound functions

Basic objects & classes – general

None think NULL, nil etc; distinct from unbound name

int Integer, C's long; transparently changes to arbitrarily long integers

bool True and False; subclasses of 1 and 0

float just that, C's double

complex two doubles; $1j * 1j == -1$

str Byte sequences, immutable, always interned.
Have built in “sprintf” support by overloading the % operator:

```
1 " Operation took %.5f seconds" % math.pi
2
3 _(" There are %(count)d days in %(name)") % \
4     {'name': 'December', 'count':31}
```

Basic objects & classes – containers

`tuple` immutable list; created by (expr1, expr2, expr3); parentheses optional

```
1 coords = (x, y)
2 normal = (coords[1], -coords[0])
3
4 # implicitly used for swapping
5 a, b = b, a
6
7 mixed = (1, "spam", Eggs())
```

Basic objects & classes – containers

`tuple` immutable list; created by (`expr1`, `expr2`, `expr3`); parentheses optional

`list` mutable list; created by [`expr1`, `expr2`, `expr3`]

```
1 mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9]
2 mylist.append(10)
3
4 mylist[1:3] = mylist[5:10:2]
```

Basic objects & classes – containers

tuple immutable list; created by (expr1, expr2, expr3); parentheses optional

list mutable list; created by [expr1, expr2, expr3]

dict key/value pairs without sequence

```
1 dictionary = {"hello": "hallo", "water":  
2             "wasser", 42: "zweiundvierzig"}  
3  
4 # alternative construction  
5 dictionary = dict(hello="hallo",  
6                 water="wasser")  
7  
8 print dict['hello']
```


Basic objects & classes – containers

`tuple` immutable list; created by `(expr1, expr2, expr3)`; parentheses optional

`list` mutable list; created by `[expr1, expr2, expr3]`

`dict` key/value pairs without sequence

- no built in type for plain numeric arrays

Class Example

```
1 class Cart(object):
2     """The typical shopping cart class example"""
3     def __init__(self):
4         self.clear()
5
6     def add(self, what, qty=1):
7         if what in self.quantities:
8             self.quantities[what] += qty
9         else:
10            self.quantities[what] = qty
11
12    def clear(self):
13        self.quantities = {}
```

Classes 1/2

Creation by keyword `class` and parent classes

```
1 class Jackalope(Jackrabbit , Antelope): pass
2
3 class Prime(int): pass
4
5 class SomethingNew(object): pass
6
7 # bad: summons ghosts from the past
8 class SomethingOld: pass
```

Classes 1/2

Creation by keyword `class` and parent classes

Constructor `__init__` takes `self` like every other method

```
1 def Jackalope(Jackrabbit , Antelope):  
2     def __init__(self , amount_antelope=0.5):  
3         self.amount_antelope = amount_antelope
```

Classes 1/2

Creation by keyword `class` and parent classes

Constructor `__init__` takes `self` like every other method

Attribute access as `self.attribute`, accessible from everywhere

```
1 def Jackalope(Jackrabbit , Antelope):  
2     def __init__(self , amount_antelope=0.5):  
3         self.amount_antelope = amount_antelope
```

Classes 1/2

Creation by keyword `class` and parent classes

Constructor `__init__` takes `self` like every other method

Attribute access as `self.attribute`, accessible from everywhere

private attributes are created by prepending `__` or `_`

```
1 class Animal(object):  
2     def __init__(self, father, mother):  
3         self._father = father  
4         self._mother = mother
```

Classes 1/2

Creation by keyword `class` and parent classes

Constructor `__init__` takes `self` like every other method

Attribute access as `self.attribute`, accessible from everywhere

private attributes are created by prepending `__` or `_`

property is used to implement accessors

```
1 class Animal(object):
2     def __init__(self, father, mother):
3         self._father = father
4         self._mother = mother
5
6         # read only
7         mother = property(lambda s: s._mother)
8
9         # changes are possible
10        father = property(lambda s:s._father,
11                          _set_father)
```

Classes 2/2

Operator overloading using special names:

```
a == b  __eq__(self, b)
a + b  __add__(self, b)
b + a  __radd__(self, a) (if a + b not
                        implemented)
a in b  __contains__(self, b)
a is b  not overloadable; checks for identity
```

Destructor `__del__`

Iterable objects define a `__iter__` function

String representations are set by `__str__` for string conversion and `__repr__` for the “look” of an object

Variables / Name resolution

Names are bound to objects inside blocks by assignment

Blocks are modules, functions and classes

Modules' bound names are called global

Classes' bound names are the class attributes (distinct from objects' attributes)

Functions' bound names are called local

Objects' attributes don't form namespaces (thus `self`)

Name resolution is static to a name

Name binding example

Works as expected:

```
1 def generate_function():
2     list = []
3     def function():
4         list.append(0)
5         print list
6     return function
```

Might not do what one expects:

```
1 def generate_3_functions():
2     result = []
3     for x in range(3):
4         list = [x]
5         def function():
6             list.append(0)
7             print list
8         result.append(function)
9     return result
```

Iterables

- ▶ Generalized list concept, generally lazy
- ▶ Typically used in for loops

```
1 for pet in ("cat", "dog", "mouse"):  
2     print pet  
3  
4 for i in range(10):  
5     print i
```

Iterables

- ▶ Generalized list concept, generally lazy
- ▶ Typically used in for loops
- ▶ Functions can be lazy iterables by using yield

```
1 def primes():
2     i = 1
3     while True:
4         i = i + 1
5         for j in range(2, i):
6             if i % j == 0:
7                 continue
8         yield i
```

Iterables

- ▶ Generalized list concept, generally lazy
- ▶ Typically used in for loops
- ▶ Functions can be lazy iterables by using yield
- ▶ Generator expressions are inline iterables

```
1 sumsquares = sum((y-f(x))**2 for x,y in \
2               zip(x_values , y_values))
```

Iterables

- ▶ Generalized list concept, generally lazy
- ▶ Typically used in for loops
- ▶ Functions can be lazy iterables by using yield
- ▶ Generator expressions are inline iterables
- ▶ List comprehensions are eager generators

```
1 cities = [City(z) for z in zip_codes if z>50]
2 print "There are %d such cities , 5th is %s"%\
3       (len(cities), cities [4])
```

Iterables

- ▶ Generalized list concept, generally lazy
- ▶ Typically used in `for` loops
- ▶ Functions can be lazy iterables by using `yield`
- ▶ Generator expressions are inline iterables
- ▶ List comprehensions are eager generators
- ▶ Arguments can be passed inside generators to create coroutines

Exceptions

- ▶ Everything not perfectly normal raises an Exception
- ▶ Hardly any “fail return values”
- ▶ Language designed with EAFP in mind
- ▶ Even syntax errors and `sys.exit` are catchable exceptions
- ▶ Subclass based catching
- ▶ Raised by `raise ValueError("No negative numbers allowed")`
- ▶ Caught by `try: ...
except SomeExceptionClass: ...`
- ▶ Can be re-raised by plain `raise`

Standard Library

Builtin namespace offers just a few functions

`sys` `exit()`, `version`, `argv`

`os` `stat()`, `environ`, `uname()`

`math` `sin()`, `e`, `sqrt()`

`itertools`, `functools` iterator and function utilities

`datetime` date and time functions

`email` email and MIME handling

`urllib2` fetch URLs

`optparse` parse command line arguments

Extending Python

- ▶ Wrapping C functions
 - ▶ ctypes
 - ▶ C API
 - ▶ Pyrex/Cython
 - ▶ automatic tools (pygtk-codegen, SWIG)
- ▶ Extending Python
 - ▶ direct access to language internals (“goto” module)
 - ▶ hack the source
 - ▶ write a PEP
 - ▶ many modules written in plain Python

Introduction

Structure, Language & Syntax

Strengths & Weaknesses

Weaknesses

(excluding fixed)

- ▶ Global Interpreter Lock
- ▶ Execution speed
- ▶ No syntax for `system()`, `regexps` etc
- ▶ Too big for small embedded systems
- ▶ Parts of the standard library unorganized (`os.stat`, `os.path.exists`, `shellutils.copy`, `os.mkdir`, `os.makedirs`)
- ▶ No `while foo = bar(): ... loops`
- ▶ No `do ... while expr loops`
- ▶ Implicit `self` in methods

Strengths

- ▶ Clear syntax
- ▶ Powerful exception handling
- ▶ List / generator operations
- ▶ Large standard library ("batteries included")
- ▶ Bindings for almost everything
- ▶ Easy to write platform independent code
- ▶ Full access to language internals

Further reading

- ▶ Python docs at <http://docs.python.org/>
- ▶ Module of the Week at <http://www.doughellmann.com/PyMOTW/>
- ▶ PEPs at <http://www.python.org/dev/peps/>, esp. 8 (Style)

Recommended packages:

- ▶ `python` — Python interpreter and standard library
- ▶ `python-doc` — Python documentation (in contrib)
- ▶ `ipython` — More powerful interactive shell