PYTHON PROGRAMMING LANGUAGE: FUNCTIONS

Python Programming Language

PREPARED FOR

Engineering Students

All Engineering College

(PP)
PREPARED BY: MS. SHWETA TIWARI
Published On: April, 2022

FALL SEMESTER, YEAR (IVth, 2nd)
FALL SESSION (2021-22)

Python 3000

Overview

- Last week Python 3000 was released
- Python 3000 == Python 3.0 == Py3k
- Designed to break backwards compatibility with the 2.x series to fix “language flaws”
- Goal: reduce feature duplication by removing old ways of doing things
- This is a big change and Python 2.x will continue in parallel for some years
  - An element of risk here: will it split the Python community?

Motivation, According to GVR

- "Open source needs to move or die"
  - Matz (creator of Ruby)
- To fix early, sticky design mistakes
  - e.g. classic classes, int division, print statement
- Changing times: time/space trade-off
  - e.g. str/unicode, int/long
- New paradigms come along
  - e.g. dict views, argument annotations

Benefits, according to GVR

- More predictable Unicode handling
- Smaller language
  - Makes “Python fits in your brain” more true
- TOOWTDI (There’s Only One Way To Do It -- The Zen of Python)
  - see Perl's TIMTOWTDI (*Tim Toady*) – “There Is More Than One Way To Do It”
- Common traps removed
- Fewer surprises
- Fewer exceptions

Major Breakages

- Print function: *print(a, b, file=sys.stderr)*
- Distinguish sharply btw. text and data
  - b"…" for bytes literals
  - "…" for (Unicode) str literals
- Dict keys() returns a set view [+items()/values()]
- No default <, <=, >, >= implementation
- 1/2 returns 0.5
- Library cleanup

Print is a Function

- print x, y -> print(x, y)
- print x, -> print(x, end=" ")
- print >>f, x -> print(x, file=f)

Dictionary Views

- Inspired by Java Collections Framework
- Remove .iterkeys(), .iteritems(), .itervalues()
- Change .keys(), .items(), .values()
- These return a *dict view*
  - Not an iterator
  - A lightweight object that can be iterated repeatedly
  - .keys(), .items() have set semantics
  - .values() has "collection" semantics
    - supports iteration and not much else

Default Comparison Changed

- In Python 2.x the default comparisons are overly forgiving

```
>>> 1 < "foo"
True
```

- In Py3k incomparable types raise an error

```
>>> 1 < "foo"
Traceback …
TypeError: unorderable types: int() < str()
```

- Rationale: 2.x default ordering is bogus
  - depends on type names
  - depends on addresses

All Strings are Unicode Strings

- Java-like model:
  - strings (the str type) are always Unicode
  - separate bytes type
  - must explicitly specify encoding to go between these
- Open issues:
  - implementation
    - fixed-width characters for O(1) indexing
    - maybe 3 internal widths: 1, 2, 4 byte characters
    - C API issues (many C APIs use C char* pointers)
  - optimize slicing and concatenation???
    - lots of issues, supporters, detractors

Int/Long Unification

- There is only one built-in integer type
- Its name is int
- Its implementation is like long in Python 2.x

Int Division Returns a Float

- Always!
- Same effect in 2.x with
  - from __future__ import division

- Use // for int division

Function Annotations

- P3k still uses dynamic typing
- P3K allows optional *function annotations* that can be used for informal type declarations
- You can attach a Python expression to describe
  - Each parameter in a function definition
  - The function's return value
- These are not part of Python's semantics but can be used by other programs, e.g., for a type checker

Function Annotations

- Example:

```
Def posint(n: int) -> bool:
    return n > 0
```

- The function object that posint is bound to will had an attribute named __annotation__ that will be the dictionary

```
{‘n': int,
 'return': bool}
```

- A number of use cases are identified in the PEP including type checking

example

```
>>> def posint(n: int) -> bool:
  return n > 0
>>> posint(10)
True
>>> posint.__annotations__
{'return': <class 'bool'>, 'n': <class 'int'>}
>>> int
<class 'int'>
>>> dir(posint)
['__annotations__', '__call__', '__class__',
  '__closure__', '__code__', '__defaults__',
  '__delattr__', '__dict__', '__doc__', '__eq__',
  '__format__', '__ge__', '__get__',
  '__getattribute__', '__globals__', '__gt__',
  '__hash__', '__init__', '__kwdefaults__', '__le__',
  '__lt__', '__module__', '__name__', '__ne__',
  '__new__', '__reduce__', '__reduce_ex__',
  '__repr__', '__setattr__', '__sizeof__', '__str__',
  '__subclasshook__']
```

Typring: LBYL vs EAFP

- How do you know you have a type error in a dynamic language like Python?
- LBYL is “Look Before You Leap”
  - Programmer explicitly checks types of values before processing, e.g., *isinstance(x,int)*
- EAFP is “Easier to Ask Forgiveness that Permission”
  - Let Python raise an error when there is a problem
- Which is better?

LBYL

- LBYL
  - Adds a performance hit
  - Requires extra programming
  - Can detect errors early, before you program does something stupid with side-effects
  - Good for for some personalities
  - But it doesn’t play well with *duck typing*
- EAFP
  - Maybe your errors will be noticed at an inopportune time

Nominative vs Structural

- nominative type system
  - type compatibility and equivalence determined by explicit declarations or type names
  - E.g., C, C++, Java
- Structural type system
  - type compatibility and equivalence determined by type's structure, not explicit declarations
  - e.g. Python’s duck typing
  - What counts on structure can vary – e.g. having a set of methods or attributes

Abstract Base Classes

- Py3K adds Abstract Base Classes
- You can define you own ‘abstract classes’ and specify their relationship to other classes
- So you can create an ABC and ‘register’ other classes as subclasses

```
from abc import ABCMeta
class MyABC:
    __metaclass__ = ABCMeta
MyABC.register(tuple)
```

- Which makes these return True

```
assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

Define ABCs for Duck Types

- This gives you a better way to extend the type system, if needed, to add types corresponding to duck types

The ‘2to3’ Tool

- http://svn.python.org/view/sandbox/trunk/2to3/
- Context-free source code translator
- Handles syntactic changes best
  — E.g. print; `…`; <>; except E, v:
- Handles built-ins pretty well
  — E.g. d.keys(), xrange(), apply()
- Has some inherant limitations
  - Doesn’t do type inferencing
  - Doesn’t follow variables in your code