How Python works: dictionaries & namespaces
Python: Whence and Why?
Why Python?

by Eric Raymond, published in Linux Journal (April, 2000)

I immediately tripped over the first odd feature of Python that everyone notices: the fact that whitespace (indentation) is actually significant in the language syntax.

... 

And, like most hackers on first realizing this fact, I recoiled in reflexive disgust.

... 

Oddly enough, Python's use of whitespace stopped feeling unnatural after about twenty minutes. I just indented code, pretty much as I would have done in a C program anyway, and it worked.
When you're writing working code nearly as fast as you can type and your misstep rate is near zero, it generally means you've achieved mastery of the language. But that didn't make sense, because it was still day one and I was regularly pausing to look up new language and library features!

This was my first clue that, in Python, I was actually dealing with an exceptionally good design.
Python for science and the humanities!

The free IPython notebook makes data analysis easier to record, understand and reproduce.

INTERACTIVE NOTEBOOKS: SHARING THE CODE

BY HELEN SHEN

Flying high above the Pacific Ocean, Titus Brown is taking a deep dive into his students' research code. The long journey from Michigan State University in East Lansing to a conference in Melbourne, Australia, provides the perfect chance for the bioinformatician to scrutinize his lab’s new algorithm for removing errors from RNA sequencing data.

Three years ago, Brown might have waited until he was back in his office. It is difficult to dig into other researchers' code without them being present to explain it, make changes and produce updated results. But these days, Brown can work with his lab from afar using a free, open-source software package called IPython, which helps researchers to keep a detailed lab notebook for their computational work.

Brown's students write explanatory text and intersperse it with raw code and the charts and figures that their algorithms generate. Sitting in the aeroplane with an IPython notebook downloaded to his laptop, Brown can interact with the work. He tweaks and re-runs the code, which executes directly in the document he is reading—allowing him to see instantly whether his changes are improving the algorithm. "I can go through their notebook, understand exactly what they did and modify it, explore different parameters and look at different views, " he says. "I can do this from anywhere in the world."

Designed to make data analysis easier to share and reproduce, the IPython notebook is being used increasingly by scientists who want to keep detailed records of their work, devise teaching modules and collaborate with others. Some researchers are even publishing the notebooks to back up their research papers—and Brown, among others, is pushing to use the program as a new form of interactive science publishing.

BETTER BOOKKEEPING

The IPython notebook was developed in 2011 by a team of researchers led by Fernando Pérez, a data scientist at the University of California, Berkeley, and computational physicist Brian Granger at California Polytechnic State University in San Luis Obispo. "We built it by solving problems that we ourselves had as researchers and educators, " says Pérez.
Python for the web!
Python as “glue”!
Python and performance

Numba gives you the power to speed up your applications with high performance functions written directly in Python. With a few annotations, array-oriented and math-heavy Python code can be just-in-time compiled to native machine instructions, similar in performance to C, C++ and Fortran, without having to switch languages or Python interpreters.

Numba works by generating optimized machine code using the LLVM compiler infrastructure at import time, runtime, or statically (using the included pycc tool). Numba supports compilation of Python to run on either CPU or GPU hardware, and is designed to integrate with the Python scientific software stack.

Example

```python
from numba import jit
from numpy import arange

# jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when function is called.
@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result

a = arange(9).reshape(3,3)
print(sum2d(a))
```
How Python works: dictionaries & namespaces
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one -- and preferably only one -- obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
Dictionaries
Dictionaries

What kinds of things do we care about when we learn a new data structure?

• What are dictionaries good for?
  Looking things up

• How do I make a dictionary?
  literals (also: \texttt{dict} & comprehensions)

• How do I query a dictionary?
  \texttt{dictValue[key]}
  \texttt{key in dictValue}
  (also: dictionary methods)

• How do I modify a dictionary?
  \texttt{dictValue[key] = newValue}
  \texttt{del dictValue[key]}
  (also: dictionary methods)
def tabulate(values):
    ...

    Given a list of values, returns a dictionary that maps each unique value to the number of times that value occurred in the list ...

def tabulate(values):
    ...

    Given a list of values, returns a dictionary that maps each
    unique value to the number of times that value occurred in the list
    ...

    result = {}

    for value in values:
        if value not in result:
            result[value] = 0

        result[value] += 1

    return result
from collections import defaultdict

def tabulate(values):
    ...
    Given a list of values, returns a dictionary that maps each
    unique value to the number of times that value occurred in the list
    ...

    result = defaultdict(int)

    for value in values:
        result[value] += 1

    return result
What does this program print, and why?

```python
x, y = 'a', 'b'

def f1():
    x = 1
    print x, y

def f2(y):
    x = 2
    print x, y

f1()
f2(3)
print type(x), type(y)
print x, y
```

(Your response)
How Python works:
Namespaces
Some vocabulary

A **binding** is a *runtime* pair: variable ↦ value.

A **namespace** is a *runtime* collection of bindings.

At runtime, an **assignment** *binds* a variable to a value.

At runtime, a **reference** *looks up* a variable’s value.

A **scope** is a region of text whose statements share a namespace.
```python
x, y = 'a', 'b'

def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
```

scopes
(determined by program code)
```python
x, y = 'a', 'b'

def f1():
    x = 1
    print x, y

def f2(y):
    x = 2
    print x, y

f1()
f2(3)
print type(x), type(y)
print x, y
```

Scopes (determined by program code)

Namespaces (a snapshot of program execution)
```python
x, y = 'a', 'b'

def f1():
    x = 1
    print x, y

def f2(y):
    x = 2
    print x, y

f1()
f2(3)
print type(x), type(y)
print x, y
```

**scopes**  
(determined by program code)

**namespaces**  
(a snapshot of program execution)
x, y = 'a', 'b'
def f1():
    x = 1
    print x, y
def f2(y):
    x = 2
    print x, y
f1()
f2(3)
print type(x), type(y)
print x, y
```python
def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
```

**scopes**
(determined by program code)

**namespaces**
(a snapshot of program execution)
```python
x, y = 'a', 'b'

def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

def f1():
    x = 1
    print(x, y)

f1()
f2(3)
```

Scopes (determined by program code)

Namespaces (a snapshot of program execution)
```python
x, y = 'a', 'b'

def f1():
    x = 1
    print x, y

def f2(y):
    x = 2
    print x, y

f1()
f2(3)
print type(x), type(y)
print x, y
```

**scopes**
(determined by program code)

**namespaces**
(a snapshot of program execution)
```python
x, y = 'a', 'b'

def f1():
    x = 1
    print x, y

def f2(y):
    x = 2
    print x, y

f1()
f2(3)
print type(x), type(y)
print x, y
```

**scopes**
(determined by program code)

**namespaces**
(a snapshot of program execution)
```python
x, y = 'a', 'b'
def f1():
    x = 1
    print(x, y)
def f2(y):
    x = 2
    print(x, y)
f1()
f2(3)
print(type(x), type(y))
print(x, y)
```
```python
x, y = 'a', 'b'
def f1():
    x = 1
    print x, y

def f2(y):
    x = 2
    print x, y

f1()
f2(3)
print type(x), type(y)
print x, y
```

### Scopes
(determined by program code)

### Namespaces
(a snapshot of program execution)
```python
x, y = 'a', 'b'
def f1():
    x = 1
    print(x, y)
def f2(y):
    x = 2
    print(x, y)
f1()
f2(3)
print(type(x), type(y))
print(x, y)
```

**scopes**  
(determined by program code)

**namespaces**  
(a snapshot of program execution)

**global**
- x → 'a'
- y → 'b'
- f1 →
- f2 →

**built-in**  
(type and others)

**file / module / session**
- x, y = 'a', 'b'
- def f1():
  - x = 1
  - print(x, y)
- def f2(y):
  - x = 2
  - print(x, y)
- f1()
- f2(3)
- print(type(x), type(y))
- print(x, y)
```python
x, y = 'a', 'b'
def f1():
    x = 1
    print x, y

def f2(y):
    x = 2
    print x, y
f1()
f2(3)
print type(x), type(y)
print x, y
```

defines scopes (determined by program code)

builds namespaces (a snapshot of program execution)
```python
x, y = 'a', 'b'

def f1():
    x = 1
    print x, y

def f2(y):
    x = 2
    print x, y

f1()
f2(3)
print type(x), type(y)
print x, y
```

**scopes**
(determined by program code)