Collections and File I/O

Basic Programming in Python

Sebastian Höffner Aline Vilks

Wed, 26 Apr 2017

Homework issues: scope

```
counter = 0
def count_up():
    counter = counter + 1

count_up()
print(counter)
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 5, in <module>
  File "<string>", line 3, in count_up
UnboundLocalError: local variable 'counter' referenced befo
```

Homework issues: scope

Python has several scopes, relevant to us are:

- local (inside functions)
- global (inside "modules" or for now: scripts)
- built-in (what Python itself offers: `len`, `range`, ...)

Collections and File I/O

Homework issues: scope

Homework issues: scope

Python has several scopes, relevant to us are:

- local (inside functions)
- global (inside "modules" or for now: scripts)
- built-in (what Python itself offers: `len`, `range`, ...)

If a variable name is used inside multiple scopes, "local" is the strongest scope.

Try not to overwrite built-in variables (your editor marks them colorful)!

Homework issues: scope – global solution

```
counter = 0
def count_up():
    global counter
    counter = counter + 1

count_up()
print(counter)
```

Output:

```
1
```

Homework issues: scope – global solution

```
counter = 0
def count_up():
    global counter
    counter = counter + 1

count_up()
print(counter)
```

Output:

```
1
```

It is not really cool to use variables as globals (there might be situations where it's necessary, but try to avoid it).

Better is to use a local solution.

Homework issues: scope – local solution

```
def count_up(counter):
    return counter + 1

counter = 0
counter = count_up(counter)
print(counter)
```

Output:

```
1
```

Homework issues: scope – while

```
counter = 0
def add10(number):
    while counter < 10:
        number += 1
        counter += 1
    return number

print(add10(2))
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 8, in <module>
  File "<string>", line 3, in add10
UnboundLocalError: local variable 'counter' referenced bef
```

Homework issues: scope – while

```
counter = 0
def add10(number):
    while counter < 10:
        number += 1
        counter += 1
    return number

print(add10(2))
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 8, in <module>
  File "<string>", line 3, in add10
UnboundLocalError: local variable 'counter' referenced bef
```

This won't work, but it's also probably not what was meant to work.

If you use a counter inside a function, you in general want it to be reset each time you call that function – so it should go inside.

Homework issues: scope – while

```
def add10(number):
    counter = 0
    while counter < 10:
        number += 1
        counter += 1
    return number

print(add10(2))
```

Output:

```
12
```

Homework remark: comparison with True

```
def fizz(number):
    return number % 3 == 0



if fizz(3) == True:
    print('fizz')
if fizz(3):
    print('fizz')
```

Output:

```
fizz
fizz
```

Homework remark: comparison with True

```
def fizz(number):
    return number % 3 == 0


if fizz(3) == True:
    print('fizz')
if fizz(3):
    print('fizz')
```

Output:

```
fizz
fizz
```

`==` `True` is never needed, you are always checking for True!

Recursion

```
def count_to_0(current):
    if current < 0:
        return
    print(current, end=', ')
    count_to_0(current - 1)

count_to_0(10)
```

Output:

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0,
```

Recursive addition

Consider two baskets of apples. How can we model putting all into the same basket?

Let's do a live example!

Recursive addition – solution

```
def add(left, right):
    if right == 0:
        return left
    return add(left + 1, right - 1)


print(add(5, 3))
```

Output:

```
8
```

Function arguments – positional

```
def function(arg0, arg1='default'):
    print(arg0, arg1)

function(0, 1)
function(0)
function(arg1=1, arg0=0)
```

Output:

```
0 1
0 default
0 1
```

Function arguments – positional

```
def function(arg0, arg1='default'):
    print(arg0, arg1)

function(0, 1)
function(0)
function(arg1=1, arg0=0)
```

Output:

```
0 1
0 default
0 1
```

- Positional arguments are the “normal” way we already discussed to provide variables.
- They always go first, followed by positional arguments with default values.
- A default value is always used if no other value is provided.
- It is possible to use arguments by their names – the order does not matter.

Function arguments – arbitrary argument lists

```
def function(*args):
    print(args)

function(1, 2, 3)
```

Output:

```
(1, 2, 3)
```

```
def function(*args):
    print(args)

function(1, 2, 3)
```

Output:

```
(1, 2, 3)
```

- Argument lists can not be used by their names.
- There can only be one argument list, as it would otherwise be indistinguishable to which argument list each argument belongs.
- The notation `(1, 2, 3)` is new, we will come back to it in a few slides.

Function arguments – keyword arguments

```
def function(**kwargs):
    print(kwargs)

function(arg0=0, mug='tea cup', animal='platypus')
```

Output:

```
{'arg0': 0, 'mug': 'tea cup', 'animal': 'platypus'}
```

Collections and File I/O

Function arguments – keyword arguments

- Keyword arguments can be captured using the double asterisk notation.
- They are stored in a dictionary.

Function arguments – keyword arguments

```
def function(**kwargs):
    print(kwargs)

function(arg0=0, mug='tea cup', animal='platypus')
```

Output:

```
{'arg0': 0, 'mug': 'tea cup', 'animal': 'platypus'}
```

Tuples, lists, dictionaries, sets

Python offers four basic collection types:

- Tuples: `(1, 2, 'hello', 2.3)`
- Lists: `[1, 2, 'hello', 2.4]`
- Dictionaries: `{'a': 1, 'b': 2}`
- Sets: `{1, 2, 'hello'}`

Collections and File I/O

Tuples, lists, dictionaries, sets

Tuples, lists, dictionaries, sets

Python offers four basic collection types:

- Tuples: `(1, 2, 'hello', 2.3)`
- Lists: `[1, 2, 'hello', 2.4]`
- Dictionaries: `{'a': 1, 'b': 2}`
- Sets: `{1, 2, 'hello'}`

They are all iterables.

Remember: Strings are iterables as well.

Tuples

```
my_fruits = ('apple', 'pear', 'banana')
print(my_fruits)
```

Output:

```
('apple', 'pear', 'banana')
```

Collections and File I/O

Tuples

Tuples are immutable, that means they are copied when we assign them to another variable.

Tuples are sorted the way they are created.

Tuples

```
my_fruits = ('apple', 'pear', 'banana')
print(my_fruits)
```

Output:

```
('apple', 'pear', 'banana')
```

Tuple functions: `len`

```
my_fruits = ('apple', 'pear', 'banana')
print(len(my_fruits))
```

Output:

```
3
```

Tuple functions: indexing

```
my_fruits = ('apple', 'pear', 'banana')
print(my_fruits[1])

for i in range(len(my_fruits)):
    print(my_fruits[i], end=' ')
```

Output:

```
pear
apple pear banana
```

Tuple functions: `index`

```
my_fruits = ('apple', 'pear', 'banana')
print(my_fruits.index('pear'))
```

Output:

```
1
```

Tuple functions: merging

```
my_fruits = ('apple', 'pear', 'banana')
my_fruits = my_fruits + ('strawberry', )
print(my_fruits)
```

Output:

```
('apple', 'pear', 'banana', 'strawberry')
```

Collections and File I/O

Tuple functions: merging

Tuple functions: merging

```
my_fruits = ('apple', 'pear', 'banana')
my_fruits = my_fruits + ('strawberry', )
print(my_fruits)
```

Output:

```
('apple', 'pear', 'banana', 'strawberry')
```

We can check the documentation or `help(tuple)` for more information about functions.

Lists

```
my_fruits = ['apple', 'pear', 'banana']
print(my_fruits)
```

Output:

```
['apple', 'pear', 'banana']
```

Lists are slightly different than tuples

```
my_fruits = ['apple', 'pear', 'banana']
your_fruits = my_fruits
your_fruits.append('strawberry')
print(my_fruits)
print(your_fruits)
```

Output:

```
['apple', 'pear', 'banana', 'strawberry']
['apple', 'pear', 'banana', 'strawberry']
```

Collections and File I/O

Lists are slightly different than tuples

Lists are mutable, which means that we only assign a reference to the object to our variables – references both point to the same instance. Thus, we modify both unless we explicitly "copy" the list.

They are also stored sorted.

Lists are slightly different than tuples

```
my_fruits = ['apple', 'pear', 'banana']
your_fruits = my_fruits
your_fruits.append('strawberry')
print(my_fruits)
print(your_fruits)
```

Output:

```
['apple', 'pear', 'banana', 'strawberry']
['apple', 'pear', 'banana', 'strawberry']
```

List functions: `insert`

```
fruits = ['apple', 'pear', 'banana']
fruits.insert(1, 'avocado')
print(fruits)
```

Output:

```
['apple', 'avocado', 'pear', 'banana']
```

List functions: `remove`

```
fruits = ['apple', 'pear', 'banana']
fruits.remove('pear')
print(fruits)
```

Output:

```
['apple', 'banana']
```

List functions: pop

```
fruits = ['apple', 'pear', 'banana']
last = fruits.pop()
print(fruits, last)
```

Output:

```
['apple', 'pear'] banana
```

Collections and File I/O

List functions: pop

List functions: pop

```
fruits = ['apple', 'pear', 'banana']
last = fruits.pop()
print(fruits, last)
```

Output:

```
['apple', 'pear'] banana
```

It also supports the same as tuples: `len(my_fruits)`, `my_fruits.index('pear')`, and `my_fruits + ['strawberry']`, indexing, etc.

Dictionaries

```
my_foods = {'fruit': 'apple', 'vegetable': 'carrot'}
print(my_foods)
```

Output:

```
{'fruit': 'apple', 'vegetable': 'carrot'}
```

Collections and File I/O

Dictionaries


```
my_foods = {'fruit': 'apple', 'vegetable': 'carrot'}
print(my_foods)
```

Output:

```
{'fruit': 'apple', 'vegetable': 'carrot'}
```

Dictionaries are also mutable.

In general they are stored in the order they are created but it is not guaranteed, so don't rely on it!

Dictionary functions: `keys` and `values`

```
my_foods = {'fruit': 'apple', 'vegetable': 'carrot'}
print(my_foods.keys())
print(my_foods.values())
print(my_foods.items())
```

Output:

```
dict_keys(['fruit', 'vegetable'])
dict_values(['apple', 'carrot'])
dict_items([('fruit', 'apple'), ('vegetable', 'carrot')])
```

Dictionary functions: indexing

```
my_foods = {'fruit': 'apple', 'vegetable': 'carrot'}
print(my_foods['fruit'])
```

Output:

```
apple
```

Dictionary functions: adding values

```
my_foods = {'fruit': 'apple', 'vegetable': 'carrot'}
my_foods['soup'] = 'potato'
print(my_foods)
```

Output:

```
{'fruit': 'apple', 'vegetable': 'carrot', 'soup': 'potato'}
```

Dictionary functions: `pop`

```
my_foods = {'fruit': 'apple', 'vegetable': 'carrot'}
my_foods.pop('fruit')
print(my_foods)
```

Output:

```
{'vegetable': 'carrot'}
```

Sets

```
my_meals = {'lunch', 'dinner', 'dinner'}
print(my_meals)
```

Output:

```
{'lunch', 'dinner'}
```

Sets

```
my_meals = {'lunch', 'dinner', 'dinner'}
print(my_meals)
```

Output:

```
{'lunch', 'dinner'}
```

Sets are unordered and have unique values. They also support some set operations. However, they are not that often used, although they are quite useful!

Set operations

```
food_a = {'steak', 'apple', 'cauliflower', 'broccoli'}
food_b = {'broccoli', 'cauliflower', 'bell pepper'}
print(food_a & food_b)
print(food_a | food_b)
print(food_a.difference(food_b))
```

Output:

```
{'cauliflower', 'broccoli'}
{'cauliflower', 'apple', 'steak', 'bell pepper', 'broccoli
{'apple', 'steak'}
```

Iteration over tuples, lists, and sets

```
fruits = ['apple', 'pear', 'banana']
for fruit in fruits:
    print(fruit, end=', ')
```

Output:

```
apple, pear, banana,
```

Collections and File I/O

Iteration over tuples, lists, and sets

Iteration over tuples, lists, and sets

```
fruits = ['apple', 'pear', 'banana']
for fruit in fruits:
    print(fruit, end=', ')
```

Output:

```
apple, pear, banana,
```

It works exactly the same way for tuples and sets.

Iteration over tuples, lists, and sets, with index

```
fruits = ['apple', 'pear', 'banana']
for i, fruit in enumerate(fruits):
    print(i, fruit, sep=':', end=', ')
```

Output:

```
0:apple, 1:pear, 2:banana,
```

Iteration over parts of tuples, lists, and sets

```
fruits = ['apple', 'pear', 'banana']
for fruit in fruits[1:]:
    print(fruit, end=', ')
```

Output:

```
pear, banana,
```

Iteration over dictionaries

```
food = {'start': 'soup', 'main': 'pizza',
        'dessert': 'ice cream'}
for key in food.keys():
    print(key, end=', ')
```

Output:

```
start, main, dessert,
```

Iteration over dictionaries – with values

```
food = {'start': 'soup', 'main': 'pizza',
        'dessert': 'ice cream'}
for key, value in food.items():
    print(key, value, end=', ', sep=':')
```

Output:

```
start:soup, main:pizza, dessert:ice cream,
```

Iteration over dictionaries – with values

```
food = {'start': 'soup', 'main': 'pizza',
        'dessert': 'ice cream'}
for key, value in food.items():
    print(key, value, end=', ', sep=':')
```

Output:

```
start:soup, main:pizza, dessert:ice cream,
```

For dictionaries we have to define what we want to iterate over.

By default the keys are used.

Nested collections

```
menu = {'main': ['pizza', 'pasta'],
        'dessert': [
            {'ice cream': ['chocolate', 'vanilla']},
            'mousse au chocolat'
        ]
       }
print(menu['dessert'][1])
```

Output:

```
mousse au chocolat
```

Function arguments and collections

Write a function `calculator`. It takes a list of arguments and performs the operation provided with the keyword `operation` on all numbers to return one result. Implement it for the operations `+`, `-`, `*`, `/`. If no operation (or an invalid one) is provided, return the first number.

Example inputs	Result
`calculator(1, 2, 3, operation='+')`	6
`calculator(2, 4, 8, operation='*')`	64
`calculator(3, 2, operation='-')`	1
`calculator(1, 2, 3, 4, 5, 6, 7, operation='+')`	28
`calculator(4, 2, 7)`	4
`calculator(4, 2, 7, operation='x')`	4

Function arguments and collections - solution

```
def calculator(*args, **kwargs):
    result = args[0]
    if 'operation' in kwargs.keys():
        for arg in args[1:]:
            if kwargs['operation'] == '+':
                result = result + arg
            # etc.
    return result
print(calculator(1, 2, 3, operation='+'))
```

Output:

```
6
```

I/O

IO or I/O or similar abbreviations usually stand for:

Input and Output

IO or I/O or similar abbreviations usually stand for:

Input and Output

Handled by (data) streams.

We already used one: the standard output stream (“stdout”)

Input and output

```
name = input('Who are you? ')
print('Hello ' + name + '!')
```

Output:

```
Who are you? Basti
Hello Basti!
```

Standard output stream

```
print(*objects, sep=' ', end='\n',
      file=sys.stdout, flush=False)
```

> *Print `objects` to the text stream `file`, separated by `sep` and followed by `end`. `sep`, `end` and `file`, if present, must be given as keyword arguments.*

(Python Documentation, Python Software Foundation 2017, https://docs.python.org/3/library/functions.html#print)

Collections and File I/O

Standard output stream

Standard output stream

```
print(*objects, sep=' ', end='\n',
      file=sys.stdout, flush=False)
```

Print `objects` to the text stream `file`, separated by `sep` and followed by `end`. `sep`, `end` and `file`, if present, must be given as keyword arguments.

(Python Documentation, Python Software Foundation 2017, https://docs.python.org/3/library/functions.html#print)

- You can ignore the `flush` parameter: some streams first collect data (buffering) and then write it (flush).
- We will now try another stream for `file` – the default is `sys.stdout`, which means to just print it to the terminal.

print to file

```
my_lottery_numbers = [21, 8, 19, 9, 1, 22]
with open('04_CollectionsFileIO/code/lottery.txt', \
          'w') as lottery_file:
    print(my_lottery_numbers, file=lottery_file)
```

Output:

File: `lottery.txt`

```
[21, 8, 19, 9, 1, 22]
```

Collections and File I/O

print to file

print to file

```
my_lottery_numbers = [21, 8, 19, 9, 1, 22]
with open('04_CollectionsFileIO/code/lottery.txt', \
          'w') as lottery_file:
    print(my_lottery_numbers, file=lottery_file)
```

Output:

File: `lottery.txt`

```
[21, 8, 19, 9, 1, 22]
```

File names: try to use relative file names so that everyone can use your code.

File modes: the letter after the file name is the mode. It can be one of:

- `write` (overwrites/creates)
- `read` (read-only)
- append (updates/creates)
- add + to open for read and write (e.g. `r+`)
- and some more

read/write to/from file

```
my_lottery_numbers = [21, 8, 19, 9, 1, 22]
with open('04_CollectionsFileIO/code/lottery.txt', \
          'w+') as lottery_file:
    lottery_file.write(my_lottery_numbers)
    result = lottery_file.read()
print(result)
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 4, in <module>
TypeError: write() argument must be str, not list
```

Collections and File I/O

read/write to/from file

read/write to/from file

```
my_lottery_numbers = [21, 8, 19, 9, 1, 22]
with open('04_CollectionsFileIO/code/lottery.txt', \
          'w+') as lottery_file:
    lottery_file.write(my_lottery_numbers)
    result = lottery_file.read()
print(result)
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 4, in <module>
TypeError: write() argument must be str, not list
```

You might find other sources on the internet using different notations, e.g. with “open” and “close” for files – just use `with` unless you can’t. Because `with` opens the file and automatically closes it for you!

Closing files is important because sometimes other processes are not allowed to access “used” files, even if you don’t really use them anymore.

Cast to the rescue

```
my_lottery_numbers = [21, 8, 19, 9, 1, 22]
with open('04_CollectionsFileIO/code/lottery.txt', \
          'w+') as lottery_file:
    lottery_file.write(str(my_lottery_numbers))
    result = lottery_file.read()
print(result)
```

Output:

Seeking the beginning

```
my_lottery_numbers = [21, 8, 19, 9, 1, 22]
with open('04_CollectionsFileIO/code/lottery.txt', \
          'w+') as lottery_file:
    lottery_file.write(str(my_lottery_numbers))
    lottery_file.seek(0)
    result = lottery_file.read()
print(result)
print(type(result))
```

Output:

```
[21, 8, 19, 9, 1, 22]
<class 'str'>
```

Reconstructing the list properly

```
with open('04_CollectionsFileIO/code/lottery.txt', \
          'r') as lottery_file:
    lottery_raw = lottery_file.read()
# This is black python magic! Let's discuss it
numbers = [int(num) for num in \
           lottery_raw[1:-1].split(',')]
print(numbers)
print(type(numbers))
```

Output:

```
[21, 8, 19, 9, 1, 22]
<class 'list'>
```

Reconstructing the list properly

```
with open('04_CollectionsFileIO/code/lottery.txt', \
          'r') as lottery_file:
    lottery_raw = lottery_file.read()
# This is black python magic! Let's discuss it
numbers = [int(num) for num in \
           lottery_raw[1:-1].split(',')]
print(numbers)
print(type(numbers))
```

Output:

```
[21, 8, 19, 9, 1, 22]
<class 'list'>
```

```
[int(num) for num in \
 lottery_raw[1:-1].split(',')]
```

is equivalent to (but shorter to write)

```
lottery_no_brackets = lottery_raw[1:-1]
lottery_strings = lottery_no_brackets.split(',')
lottery_numbers = []
for num in lottery_strings:
    lottery_numbers.append(int(num))
```

This is called **list comprehension**.

Reading a file line by line

```
with open('04_CollectionsFileIO/code/names.txt', \
          'r') as names_file:
    names = names_file.read().splitlines()
print(names)
```

Output:

```
['Graham Chapman', 'John Cleese', 'Terry Gilliam', 'Eric Io
```

Reading a file line by line

```
with open('04_CollectionsFileIO/code/names.txt', \
          'r') as names_file:
    lastnames = []
    for name in names_file:
        first_and_last = name.split()
        lastnames.append(first_and_last[1])

print(lastnames)
```

Output:

```
['Chapman', 'Cleese', 'Gilliam', 'Idle', 'Jones', 'Palin']
```

Your fourth homework

- Do some simple vector algebra with loops.
- Play hangman to master input and output!

The last slide

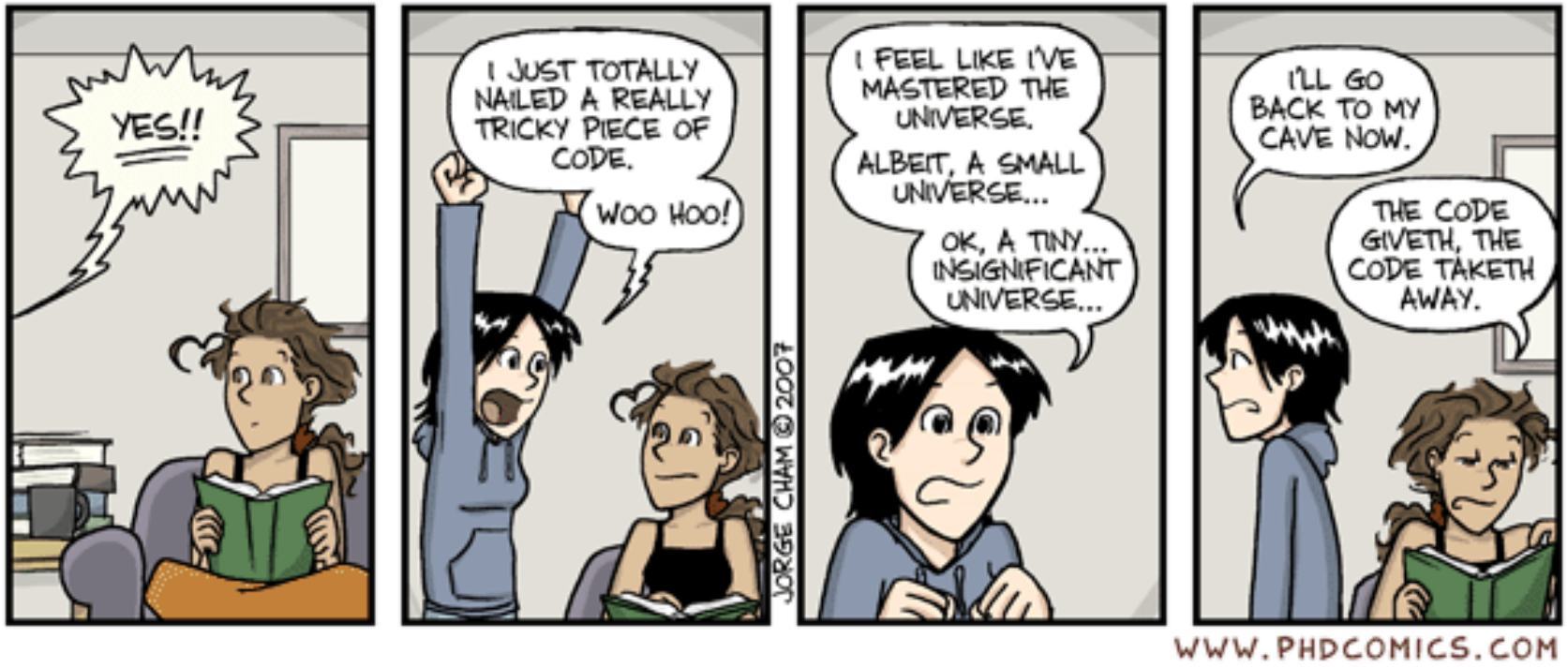

Figure 1: Master of the Universe (Cham 2007)

References

Cham, Jorge. 2007. "Master of the Universe." *PhD – Piled Higher and Deeper*, November.

Python Software Foundation. 2017. *Python 3.6.0 Documentation*. 3.6.0 ed. Beaverton, Oregon, USA: Python Software Foundation.