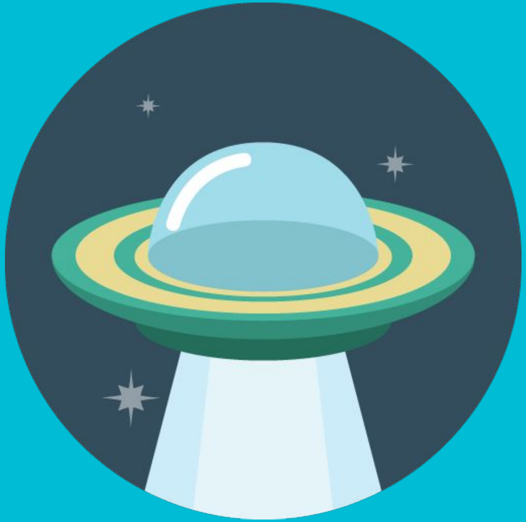
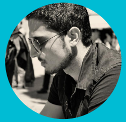
The CLOSURES that moved Spielberg





Pablo Galindo
@pablogsalgado

Víctor Terrón
@pyctor

1.

UnboundLocalError

Such a common —yet mysterious— error that even the **Python FAQ** answers it.

```
x = 42
def show_x():
    print(x)

show()

>> 42
```

This works

The usual way to fix it: "modify lines of code **more or less randomly** until the problem goes away"

```
x = 42
def show_x_plus_one():
    x += 1
    print(x)

show_x_plus_one()
```

This doesn't :-(

```
>> UnboundLocalError: local variable 'x' referenced before assignment
```

But
WHY?

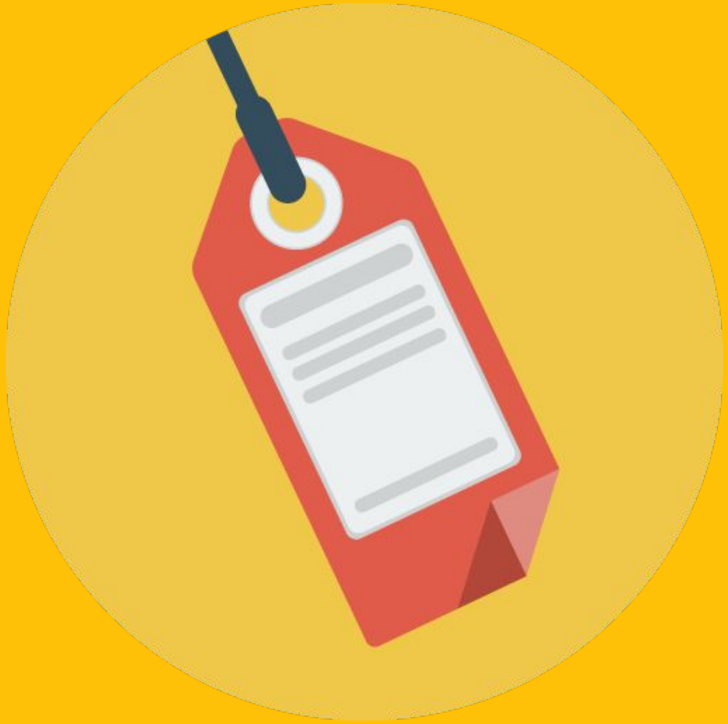
Why can we **read** x, but not **modify** it?

According to the Python FAQ:

"This is because when you make an assignment to a variable in a scope, **that variable becomes local to that scope and shadows any similarly named variable in the outer scope**. [...] Consequently when the earlier print(x) attempts to print the uninitialized local variable and an error results."

2.

What's a variable



They are more "**names**" or "**identifiers**"

```
x = 5   # x points to an integer
x = 3   # ... now to another integer
x = "a" # ... and now to a string
```

Dynamic **Typing**

Think of variables as **'labels'** or 'aliases'

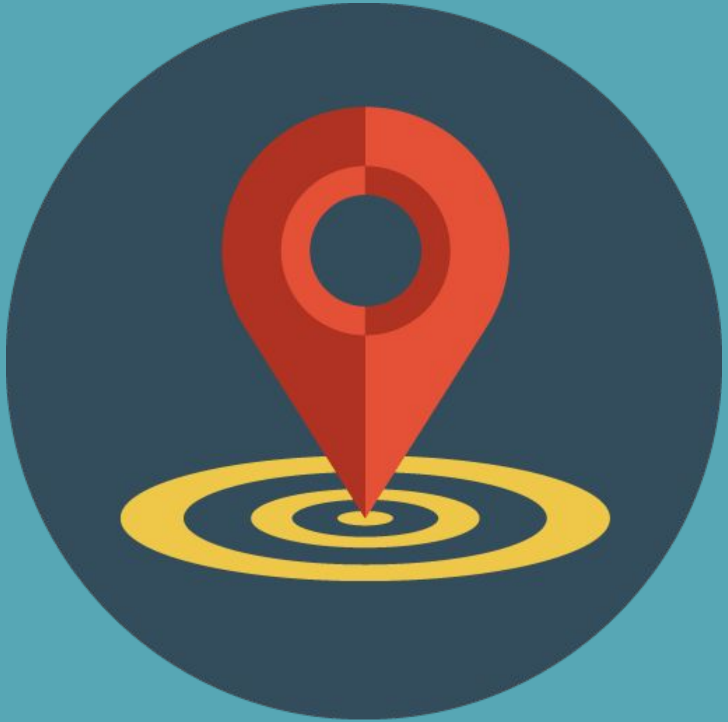
We can see it ourselves with the built-in **id()**

```
x = 42
print(id(x))    # 36115784
print(id(42))   # 36115784

x = 13          # ‘x’ points now to another object
print(id(x))    # 36114488
print(id(13))   # 36114488
```

Memory address

3.

Namespaces



Namespaces

A name-value mapping

— or "name-object", strictly speaking.

These mappings are

dictionaries

Much unexpected. Very surprise. How shock.

vars() shows us the **current** namespace:

```python
x = 4
y = 7
print(vars())
```

```
{'__loader__': <_frozen_importlib_external.SourceFileLoader
object at 0x7f56d60a6358>, 'x': 4, 'y': 7, '__spec__': None,
'__cached__': None, '__name__': '__main__', '__builtins__':
<module 'builtins' (built-in)>, '__doc__': None, '__package__':
None}
```

Probably **easier** to **understand** this way:

```
x = 4
y = 7
namespace = vars().copy()
for name, value in sorted(namespace.items()):
    print(name, "->", value)

(...)
x -> 4
y -> 7
```

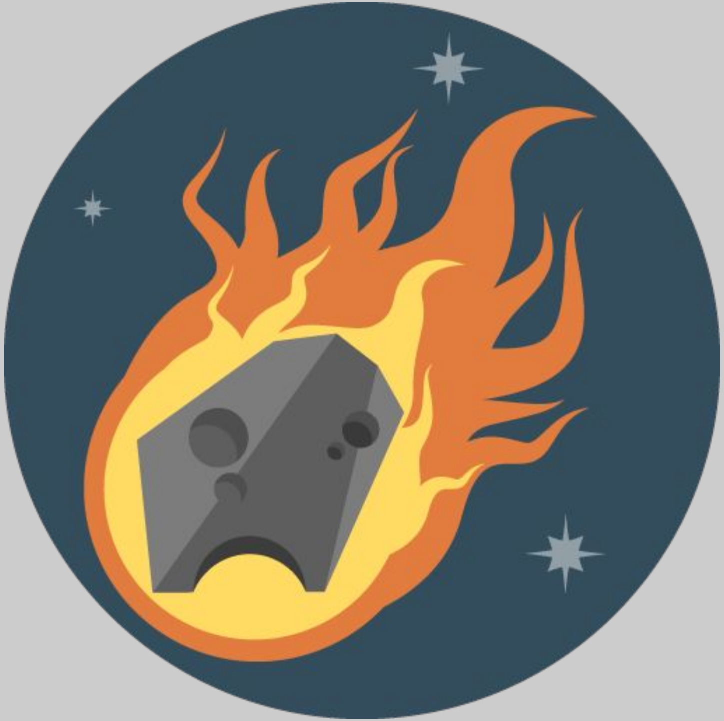
Accessing a variable is thus

a dictionary look-up

```
x = 5
print(x)                # prints 5
print(vars()['x'])  # also prints 5
```

4.

Detour – to Hell



We *could* create variables **dynamically**:

```
vars()['y'] = 8
print(y)  # prints 8
```

Please don't do this

It **doesn't** **always** work

```
def foo():
    vars()['z'] = 3
    print(z)


foo()

>> NameError: name 'z' is not defined
```

It breaks **inside** a function

Modifying the namespace is undefined

According to the Python docs:

“The contents of this dictionary **should not** be modified; changes **may not affect** the values of local and free variables used by the interpreter”

There are **multiple** namespaces

And we can use the **same** name in **different** namespaces.

Each function defines **its own** namespace.

```python
x = 5
def foo():
    x = 3
    print(x)

foo()      # prints 3
print(x)   # prints 5
```

Doesn't modify **outer** x

Let's make sure with **vars()**

```
x = 5
def foo():
    x = 3
    print("Inside foo() ->", vars()["x"])

foo()
print("Outside of foo() ->", vars()["x"])

>> Inside foo() -> 3
>> Outside of foo() -> 5
```

5.

Scopes

Namespaces are organized in a **hierarchy**

The scope defines the **point** of the hierarchy where **each namespace** is **visible.**

Name resolution basically means

"read backwards"

the source code to determine to which entity a name refers

An example that **works**

```
x = 5
def spam():
    print(x)

spam()

>> 5
```

x? What's x? I don't know!

An example that **doesn't** work

```
x = 5
def spam():
    y = 6
    print(y)

print(y)
```

y only exists **inside** spam()

```
>> NameError: name 'y' is not defined
```

Variable **shadowing**

```
x = 3
def spam():
    x = 7
    print(x)

spam()

>> 7
```

Two variables named `x`

So what's the difference?

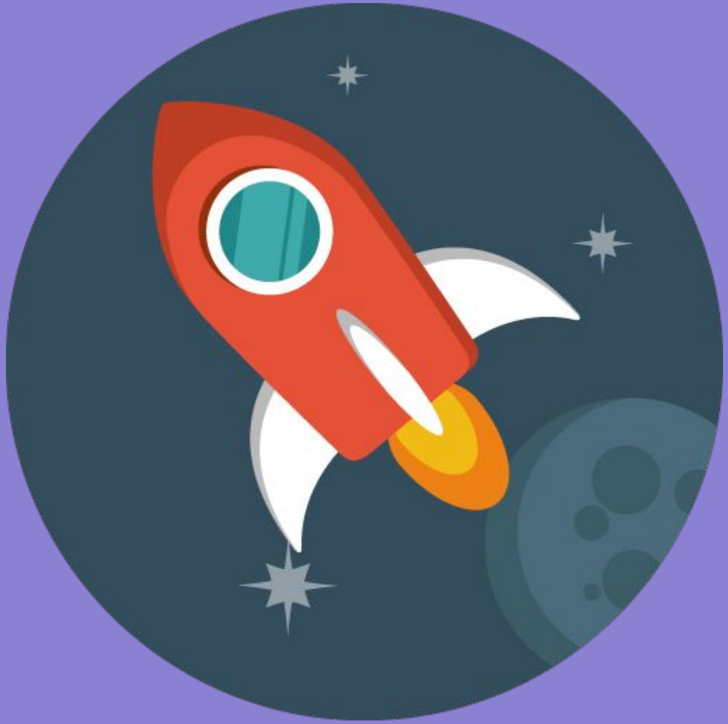
- A **namespace** maps names to objects.
- A **scope** defines **which namespaces** will be looked at and **in what order**.

In practice, we can use them

interchangeably

6.

Bounded vs Free



A **bounded variable** doesn't depend on the **context** of the function call.

```
x = 2
def spam(x):
    print(x)

spam(2)

>> 2
```

Bounded variable

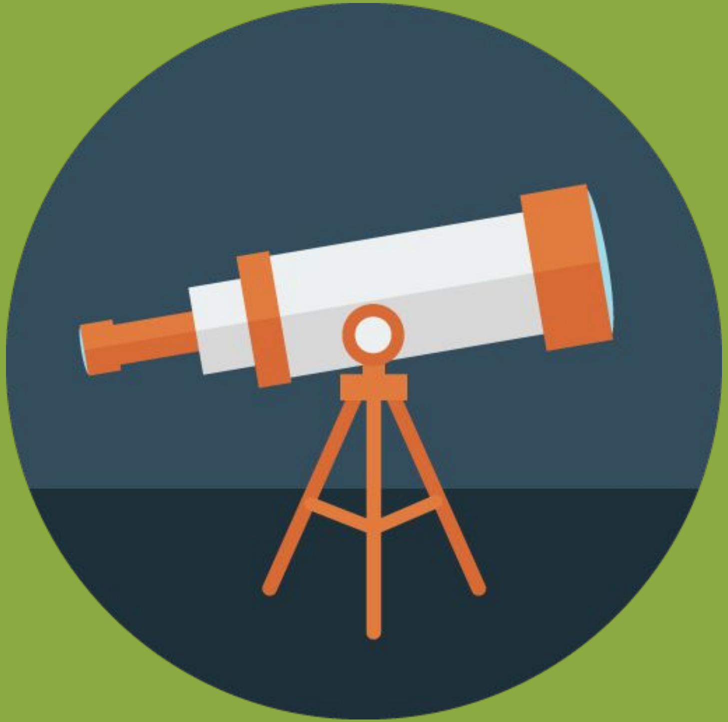
```python
a = 3
def foo():
    print(a)

foo()

>> 3
```

Free variable

A **free variable** is determined **dynamically** at run time searching the name of the variable **backwards** on the **function call stack**.

7.

Back to UnboundLocalError



```python
x = 42
def show_x_plus_one():
    x += 1
    print(x)
```

```python
x = 42
def show_x_plus_one():
    x = x + 1
    print(x)
```

x? What x?

Remember [the Python FAQ] wise words:

"… when you make an assignment to a variable in a scope, **that variable becomes local to that scope and shadows any similarly named variable in the outer scope**"

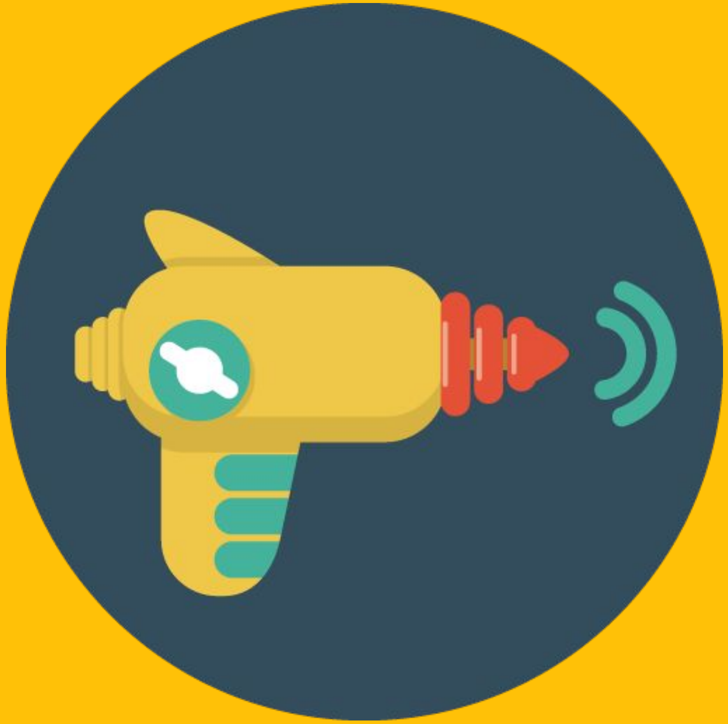
Use **global** to access the **outer scope**

```python
x = 42
def foo():
    global x
    x += 1
    print(x)
```

x is now the **outer** x

8.

Nested functions



Python functions are first-class objects

According to Wikipedia:

"A **first class object** is an entity that can be dynamically created, destroyed, passed to a function, returned as a value, and **have all the rights as other variables** in the programming language have"

Assign a function to **variable**

```
def say_spam():
    print("spam!")

cry = say_spam
cry()

>> spam!
```

No parentheses!

Define a function **inside** another

```
def welcome():
    def say_hello():
        return ”Hello, ”
    print(say_hello(), name)

welcome(”Íñigo Montoya”)

>> Hello, Íñigo Montoya
```

Pass a function as an **argument**

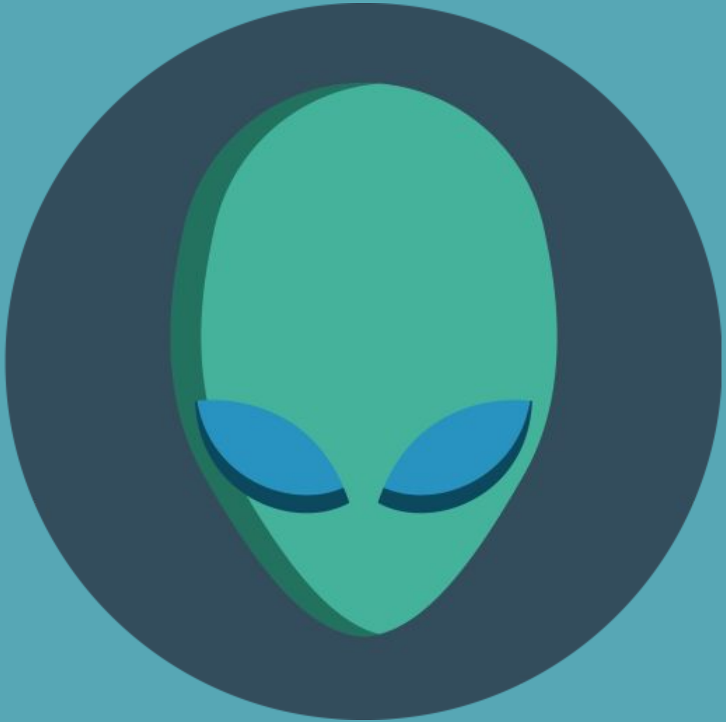
```
def solve(func, values):
    return func(values)

solve(sum, [1, 2, 3]))

>> 8
```

Create and **return** functions

```
def make_adder(constant):
    def adder(x):
        return constant + x
    return adder

add_two = make_adder(2)
print(add_two(3))
print(add_two(9))

>> 5
>> 11
```

9.

Closures



```python
def make_adder(constant):
    def adder(x):
        return constant + x
    return adder

add_two = make_adder(2)
add_two(4)

>> 6
```

add_two() **remembers** the value of **constant**

```python
def foo(x):
    y = x + 3
    print(y)

foo(1)
print(y)

>> 4
>> NameError: name 'y' is not defined
```

y no longer **exists**

Python knows we'll need **constant**...

```
def make_adder(constant):
    def adder(x):
        return constant + x
    return adder

add_two = make_adder(2)
add_two(3)

>> 5
```

Namespace of make_adder()
no longer **exists**

We remember the **original** value

```
def make_adder(constant):
    def adder(x):
        return constant + x
    return adder

constant = 2
add_two = make_adder(constant)
constant = 7
add_two(3)

>> 5
```

We say that `adder()` is

closed over

the variable `'constant'`

Hence the name '**closures**'

Not everything is a **closure**

```python
x = 42
def foo():
    print(x)
```

Not a closure

Everybody can read **global** variables.

No need for **wizardry** here.

There are three levels

global

non-local

local

It's only a closure if we remember a non-local

I.e., closures only happen with **nested functions**

10.

Closures: HowTo

Nested

Functions

— or "I live inside you, man"

We want to find the **better** speaker?

```
def better_person():
    people = {'Victor Terron' : 4,
              'Pablo Galindo' : float('inf')  }

    ...
```

max()

Very maximum. Much elements. Wow.

How we can make a **key** function?

```
def better_person():
    people = {'Victor Terron' : 4,
              'Pablo Galindo' : float('inf')  }
    return max(people.keys(), key = ??????)
```

Function with only
one argument here

How we can make a **key** function?

```
def better_person():
    people = {'Victor Terron' : 4,
              'Pablo Galindo' : float('inf')  }

    def score( name ):
        return people[name]
    return max(people.keys(), key = score)
```

Nested function

How we can make a **key** function?

```
def better_person():
    people = {'Victor Terron' : 4,
              'Pablo Galindo' : float('inf')  }

    def score( name ):
        return people[name]
    return max(people.keys(), key = score)
```

I am a closure!
Notice me, Senpai!

The **correct way** of doing this

```
def better_person():
    people = {'Victor Terron' : 4,
              'Pablo Galindo' : float('inf')  }
    return max(people.keys(), key = people.get)
```

Malicious

Closures

— or "These damn closures are gonna kill me"

A prime generator

A prime generator

```
import itertools
```

People panic!

A prime generator

```
import itertools

def prime_generator():
        numbers = itertools.count(2)
```

next(numbers) -> 2, 3, 4...

A prime generator

```
import itertools

def prime_generator():
        numbers = itertools.count(2)
        while True:
            next_prime = next(numbers)
            yield next_prime
```

This will yield 2

A prime generator

```python
import itertools

def prime_generator():
        numbers = itertools.count(2)
        while True:
            next_prime = next(numbers)
            yield next_prime
            numbers = filter(lambda n: n % next_prime, numbers)
```

3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~,...

A prime generator

```
import itertools

def prime_generator():
        numbers = itertools.count(2)
        while True:
            next_prime = next(numbers)
            yield next_prime
            numbers = filter(lambda n: n % next_prime, numbers)
```

This will yield **3**

A prime generator

```
import itertools

def prime_generator():
        numbers = itertools.count(2)
        while True:
            next_prime = next(numbers)
            yield next_prime
            numbers = filter(lambda n: n % next_prime, numbers)
```

5, 7, ~~9~~, 11, 13, ~~15~~, 17…

closures
gonna
closure

```
>>> next(prime_generator())
```

1,2,3,4,5,6,7,...

A prime **generator**

```
import itertools

def prime_generator():
        numbers = itertools.count(2)
        while True:
            next_prime = next(numbers)
            yield next_prime
            numbers = filter(lambda n: n % next_prime, numbers)
```

I am a closure!

We remember the **original** value

```
def make_adder(constant):
    def adder(x):
        return constant + x
    return adder

constant = 2
add_two = make_adder(constant)
constant = 7
add_two(3)

>> 5
```

A prime generator

```
import itertools

def prime_generator():
        numbers = itertools.count(2)
        while True:
            next_prime = next(numbers)
            yield next_prime
            numbers = filter(lambda n: n % next_prime, numbers)
```

I am a closure!

A prime generator

```
import itertools

def prime_generator():
        numbers = itertools.count(2)
        while True:
            next_prime = next(numbers)
            yield next_prime
            numbers = filter(lambda n, prime = next_prime: n % prime, numbers)
```

A prime generator

```
import itertools

def prime_generator():
        numbers = itertools.count(2)
        while True:
            next_prime = next(numbers)
            yield next_prime
            numbers = filter(lambda n, prime = next_prime: n % prime, numbers)
```

```
>>> next(prime_generator())
```

2,3,5,7,11,13,...

Capturing

“future” variables

— or "I did not know I could do that!"

A context manager

```
with open('my_file') as fd:
      print(fd.readlines())

(...)
```

A context manager

1

```
with open('my_file') as fd:
      print(fd.readlines())

(...)

@contextmanager
def open(file_path):
     fd = os.open(file_path)
```

A context manager

1

```
with open('my_file') as fd:
     print(fd.readlines())

(...)

@contextmanager
def open(file_path):
     fd = os.open(file_path)
     yield fd
```

2

A context manager

1

```
with open('my_file') as fd:
     print(fd.readlines())

(...)

@contextmanager
def open(file_path):
     fd = os.open(file_path)
     yield fd
```

3

2

A context manager

```
with open('my_file') as fd:
     print(fd.readlines())


(...)


@contextmanager
def open(file_path):
    fd = os.open(file_path)
    yield fd
    fd.close()
```

A timer context manager

```
@contextmanager
def measure_time():
    t0 = time.time()
    def timer():
        return t1 - t0
    yield timer
    t1 = time.time()
```

A timer context manager

```
@contextmanager
def measure_time():
    t0 = time.time()
    def timer():
        return t1 - t0
    yield timer
    t1 = time.time()
```

I am from the future!

A timer context manager

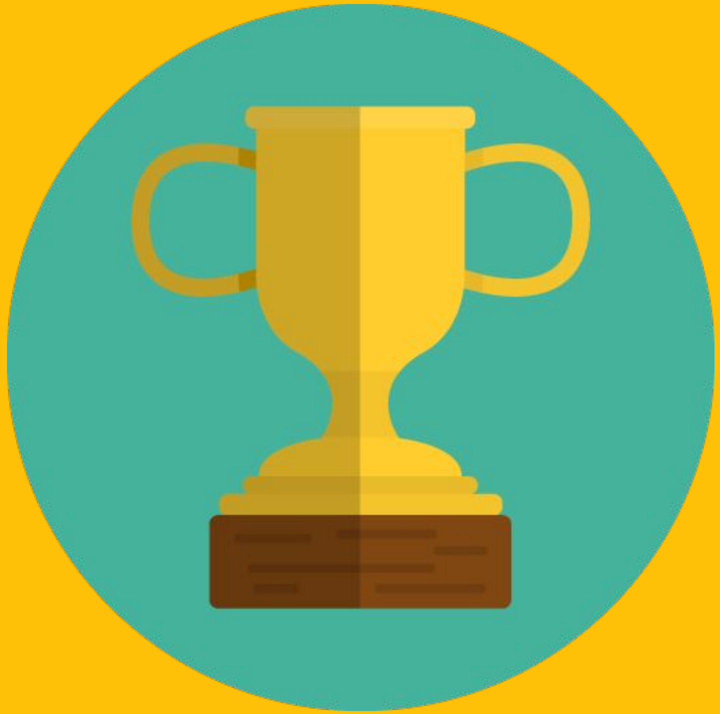
```
@contextmanager
def measure_time():
    t0 = time.time()
    def timer():
        return t1 - t0
    yield timer
    t1 = time.time()
```

```
with measure_time() as timer:
    [x for x in range(100000000)]

print(timer())
>>> 0.345643422323
```

8.

Code optimization*

*Terms and conditions may apply



```python
import math
def stupid_operation():
    with measure_time() as timer:
        for i in range(10000000):
            math.sqrt(i)
    print(timer())


stupid_operation()
>>> 2.1278745144
```

```python
import math
def stupid_operation():
    sqrt = math.sqrt
    with measure_time() as timer:
        for i in range(10000000):
            sqrt(i)
    print(timer())


stupid_operation()
```

```python
import math
def stupid_operation():
    sqrt = math.sqrt
    with measure_time() as timer:
        for i in range(10000000):
            sqrt(i)
    print(timer())


stupid_operation()
```

sqrt is local here!

```python
import math
def stupid_operation():
    sqrt = math.sqrt
    with measure_time() as timer:
        for i in range(10000000):
            sqrt(i)
    print(timer())


stupid_operation()
>>> 1.0354278144
```

2x times faster

Whoa! That’s a big number, aren’t you amazed?

MAKE_DRAMA() *

* No closures were harmed in the making of this representation.

```python
import math
def stupid_operation():
    with measure_time() as timer:
        for i in range(10000000):
            math.sqrt(i)
    print(timer())


stupid_operation()
>>> 2.1278745144
```

```python
import math
def stupid_operation():
    sqrt = math.sqrt
    with measure_time() as timer:
        for i in range(10000000):
            sqrt(i)
    print(timer())


stupid_operation()
>>> 1.0354278144
```

Using **a method** inside a **crazy loop**

```
import numpy
numbers = numpy.array([1,2,3,4,5,6,7,8,9])

for i in range(10000000):
    numbers.sum()
```

This dot wants to suck your soul

Global functions are EXPENSIVE

“In CPython, global variables and functions (including package imports) **are much more expensive** to reference than locals; avoid them.”

http://pypy.org/performance.html

```python
def nlargest(n, iterable):
    """Find the n largest elements in a dataset.

    Equivalent to:  sorted(iterable, reverse=True)[:n]
    """
    it = iter(iterable)
    result = list(islice(it, n))
    if not result:
        return result
    heapify(result)
    _heappushpop = heappushpop
    for elem in it:
        _heappushpop(result, elem)
    result.sort(reverse=True)
    return result
```

"The only thing missing from Jurassic Park were **closures**" *

– *Steven Spielberg*

*This may or may not have happened

Image by Gerald Geronimo (CC)

Recommended **resources**

Code Like a Pythonista: Idiomatic Python:
http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html

Relationship between scope and namespaces:
http://programmers.stackexchange.com/q/273302

Understanding UnboundLocalError in Python:
http://eli.thegreenplace.net/2011/05/15/understanding-unboundlocalerror-in-python

Finding Closures with Closures:
https://www.youtube.com/watch?v=E9wS6LdXM8Y