Closures in Python

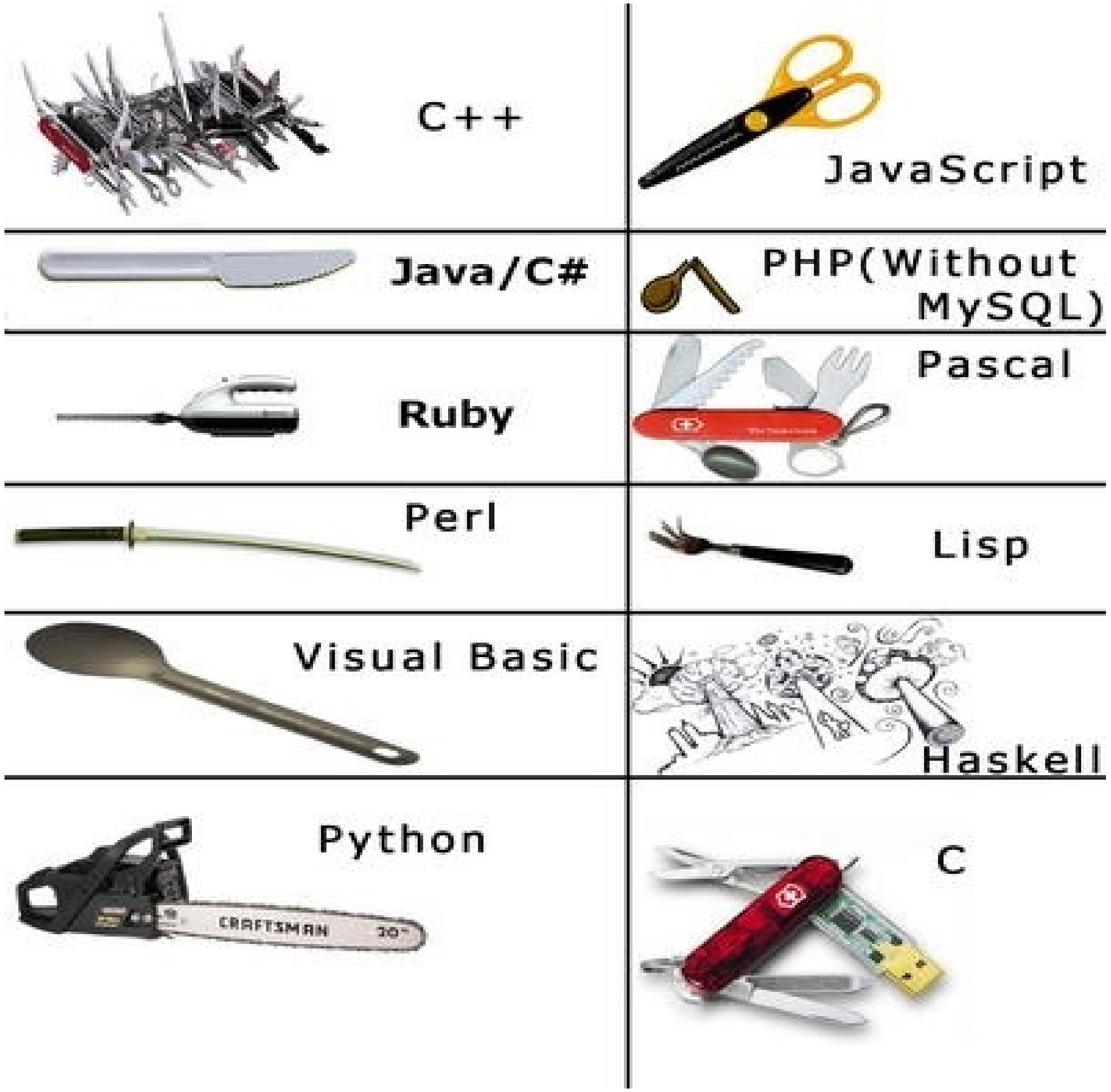
Slavomír Hudák

pycon.sk @ 10.3.2017

Agenda

- Scopes
- Function as first class citizen
- Lambdas
- Closures
- Examples
- Free variables, GC, closures over functions
- Quiz

Which languages do you use?



Initial question

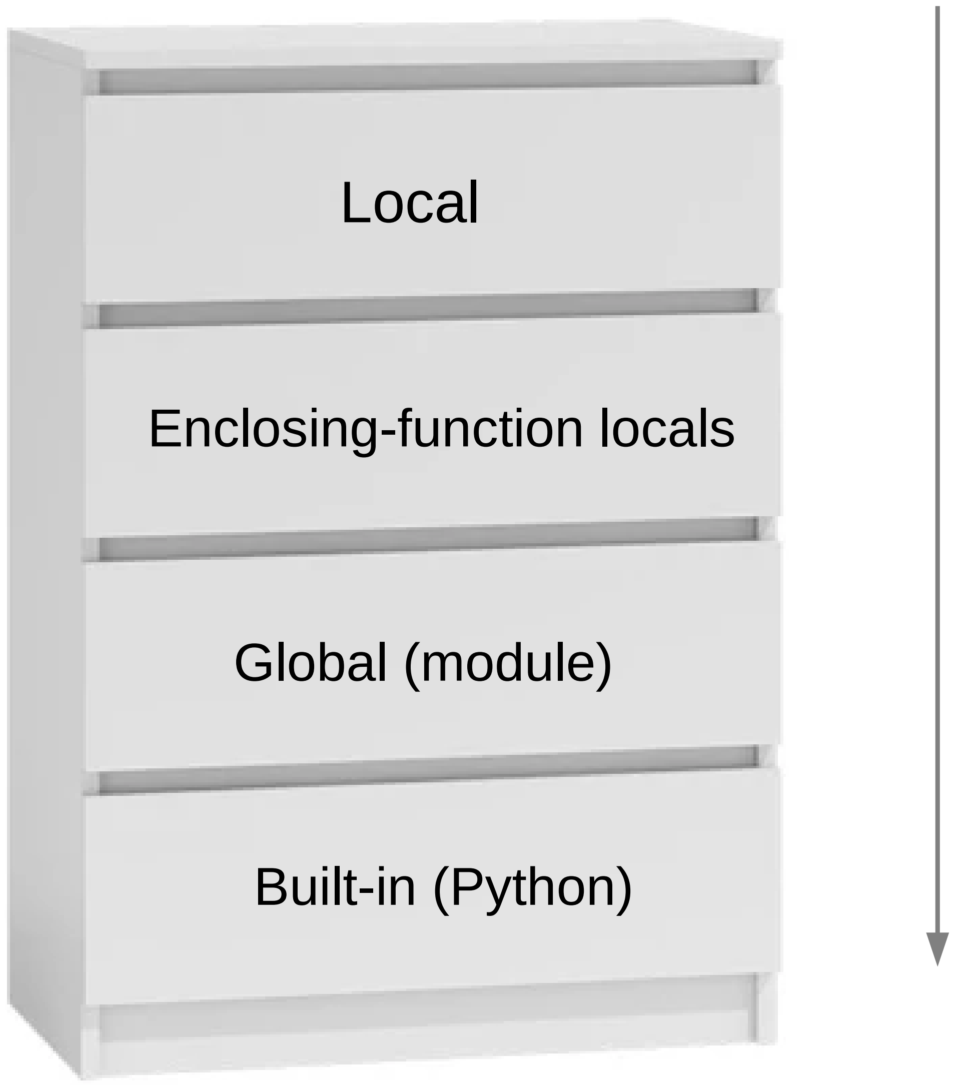
```
# This code has one problem

funcs = []
for i in range(0, 10):
    funcs.append(lambda: print(i))

for f in funcs:
    f()
```

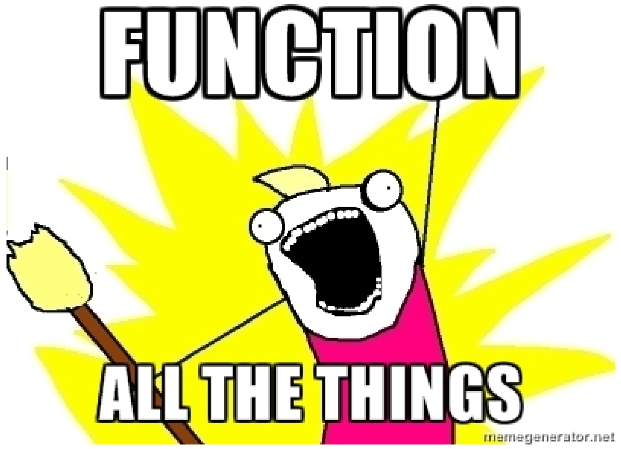
Scopes

Scopes


Local
Enclosing-function locals
Global (module)
Built-in (Python)

Functions as first class citizen

- Can be passed as argument
- Can be assigned to a variable
- Can be returned from a function
- … like you would work with data



Lambdas

- lambda x,y: x+y
- Limited anonymous functions
  - Single expression only
- Define where needed
- Immediately call if needed
  - (lambda x: x*x)(5)

Classes, Objects and State

```
class PyconPrinter():
    def __init__(self, prefix="pycon:"):
        self.prefix = prefix
    def __call__(self, msg):
        print(self.prefix, msg)

p = PyconPrinter()
p("Hi all from regular object") # pycon: Hi all from regular object
```

Closures

```
# closure
def pycon_printer(prefix="pycon:"):
    def pycon_print(msg):
        print(prefix, msg)
    return pycon_print

p = pycon_printer()
p("Hi all from closure") # pycon: Hi all from closure
```

Closures

```
# closure
def pycon_printer(prefix="pycon:"):
    def pycon_print(msg):
        print(prefix, msg)
    return pycon_print

p = pycon_printer()
p("Hi all from closure") # pycon: Hi all from closure
```

prefix is in outer scope,
After return becomes “closed” (free) variable

Closures

```
# closure
def pycon_printer(prefix="pycon:"):
    def pycon_print(msg):
        print(prefix, msg)
    return pycon_print

p = pycon_printer()
p("Hi all from closure") # pycon: Hi all from closure
```

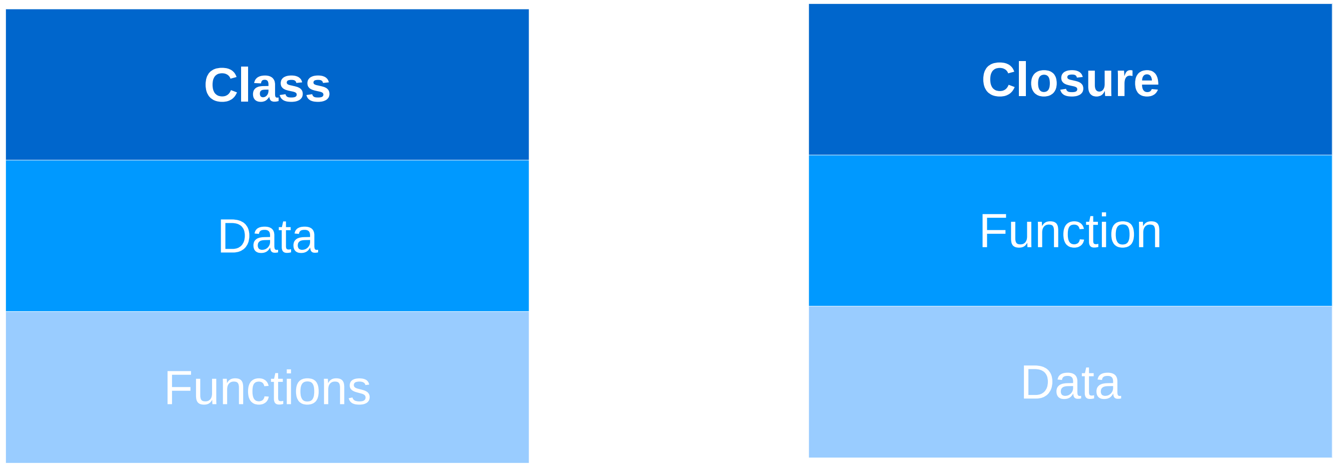
Inner function becomes closure after return and closes over state stored in prefix

Closures (continued)

- Closure is a function that can access data of the scope it was created in
- Typically used as:
  - Simple logic for other API without need to create wrapper (filter(lambda x: x>5, list)
  - Event handlers
  - Template method & other design patterns (GoF)
  - Poor man’s objects
  - Elegant solution when required single function with some extra state

Closures vs Objects (Class)

- Closure is different way of looking at an object



Closures (continued)

- When used inappropriately
  - Memory leaks (holding references to large objects you wanted to dispose and preventing garbage collection)
  - Unclear, difficult to read code by your colleagues or community

Closures - Examples

```
# average closure - variant with list
def create_avg():
    items = []
    def add(num):
        items.append(num)
        print(sum(items)/len(items))
    return add

avg = create_avg()
avg(4) # 4.0
avg(5) # 4.5
avg(6) # 5.0
```

Closures - Examples

```
# average closure - variant with list
def create_avg():
    items = []
    def add(num):
        items.append(num)
        print(sum(items)/len(items))
    return add

avg = create_avg()
avg(4) # 4.0
avg(5) # 4.5
avg(6) # 5.0
```

Closures - Examples

```
def counter():
    count = 0
    def inc_count():
        # increase counter
        count += 1
        print("Called", count, "times")
    return inc_count

inc = counter()
inc() # throws error
```

Closures - Examples

```
def counter():
    count = 0
    def inc_count():
        nonlocal count
        count += 1
        print("Called", count, "times")
    return inc_count

inc = counter()
inc() # Called 1 times
inc() # Called 2 times
inc() # Called 3 times
```

1) Read from count and add 1
2) Store result in count => conflict
   python wants to create local variable.

   Solution: specify count as nonlocal

Where are free variables stored?

```python
def create_condition(limit):
    return lambda item: item > limit

c = create_condition(5)

print(c.__closure__)
# (<cell at 0x7fc301eeb708: int object at 0xa68ac0>,)
print(c.__closure__[0].cell_contents)
# 5
```

Closure over function

```python
# closure over function
def counter(func):
    count = 0
    def inc_count():
        func()
        nonlocal count
        count += 1
        print("    Called", count, "times")
    return inc_count

def hello():
    print("Hi PyCon")

inc = counter(hello)
inc()
inc()
inc()
```

```
OUTPUT

Hi PyCon
    Called 1 times
Hi PyCon
    Called 2 times
Hi PyCon
    Called 3 times
```

Closure over func – composition example

```python
def create_logic(func):
    def when(a,b):
        if not (callable(a) and callable(b)):
            raise TypeError("Expecting callable for input parameters")
        return func(a,b)
    return when

_and = create_logic(lambda x,y: x() and y())
_or = create_logic(lambda x,y: x() or y())
```

Closure over func – composition example

```
# imagine these are some real validators
validate1 = lambda: True
validate2 = lambda: False

print(_and(validate1, validate1)) # True
print(_and(validate2, validate1)) # False
```

Closure over func – composition example

```
# more complex composition
is_valid = _and(
   lambda: _or(validate1, validate2),
   lambda: _and(validate1, validate1)
   )

#       and
#        / \
#    or    and
#    / \      / \
# v1 v2 v1 v1

print(is_valid) # True
```

Decorators

```python
def counter(func):
    count = 0
    def inc_count(*args, **kwargs):
        print("Calling", func.__name__)
        func(*args, **kwargs)
        nonlocal count
        count += 1
        print("        Called", count, "times")
    return inc_count

@counter
def follow():
    print("    Follow Freeman")

follow()
follow()
follow()
```

```
OUTPUT

Calling follow
    Follow Freeman
        Called 1 times
Calling follow
    Follow Freeman
        Called 2 times
Calling follow
    Follow Freeman
        Called 3 times
```

Closures – Examples (Closure vs Class)

```python
def create_filter(threshold):
    def filter_it(iterable):
        return [x for x in iterable if x > threshold]
    return filter_it
```

```python
class Filter:
    def __init__(self, threshold):
        self.threshold = threshold
    def __call__(self, iterable):
        return [x for x in iterable if x > self.threshold]
```

Closures – Examples (Closure vs Class v2)

```
def create_filter(threshold):
    return lambda iterable: [x for x in iterable if x > threshold]
```

```
class Filter:
    def __init__(self, threshold):
        self.threshold = threshold
    def __call__(self, iterable):
        return [x for x in iterable if x > self.threshold]
```

Quiz (1)

```
# does this code throw ?

def outer():
    var = 1

    def inner():
        var += 1

    return inner
```

Quiz (2)

```
# how to fix the below code?

funcs = []
for i in range(0, 10):
    funcs.append(lambda: print(i))

for f in funcs:
    f()
```

Quiz (2)

```
# variant 1

funcs = []
for i in range(0, 10):
    def outer(x):
        return lambda: print(x)
    funcs.append(outer(i))

for f in funcs:
    f()
```

Quiz (2)

```
# variant 2

funcs = []
for i in range(0, 10):
    funcs.append((lambda x: lambda: print(x))(i))

for f in funcs:
    f()
```

Quiz (2)

```
# variant 3

funcs = []
for i in range(0, 10):
    funcs.append(lambda x=i: print(x))

for f in funcs:
    f()
```

Quiz (2)

```
# variant 4
funcs = [(lambda x: lambda: print(x))(x) for x  in range(1,10)]


# variant 5
class Func:
    def __init__(self, i): self.i = i
    def __call__(self): print(self.i)

funcs = [Func(i) for i in range(1,10)]
```

Get in touch

Sample source codes

https://github.com/besnik/pycon2017-closures

Contact

https://www.linkedin.com/in/slafco/

https://twitter.com/besnikgeek

Thank you