ThoughtWorks®

the meaning of @fun

DECORATORS DECODED

A gentle introduction to Python metaprogramming

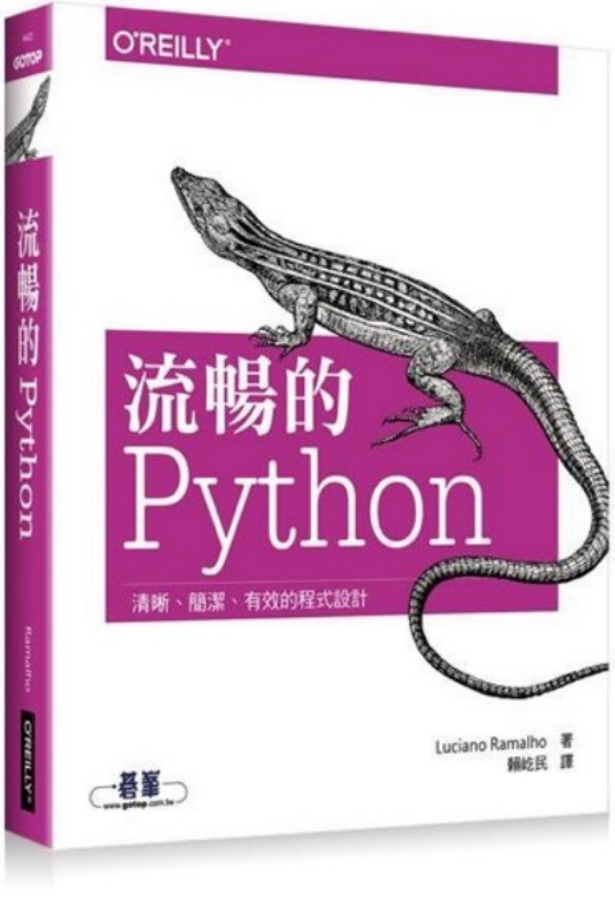
ThoughtWorks®

LUCIANO RAMALHO

Technical Principal

@ramalhoorg
luciano.ramalho@thoughtworks.com

FLUENT PYTHON, MY FIRST BOOK



Fluent Python (O'Reilly, 2015)

Python Fluente (Novatec, 2015)

Python к вершинам мастерства* (DMK, 2015)

流暢的 Python† (Gotop, 2016)

also in **Polish, Korean...**

** Python. To the heights of excellence*
† Smooth Python

ThoughtWorks®

OVERVIEW

What we'll cover

TOPICS

- Review: functions as objects
- Introducing decorators
- Registration decorators
- Closures
- Decorators that affect behavior
- Parametrized decorators
- Class-based decorators

ThoughtWorks®

FUNCTIONS AS OBJECTS

Naturally

I have never considered Python to be heavily influenced by functional languages, no matter what people say or think.

I was much more familiar with imperative languages such as C and Algol 68 and although I had made functions first-class objects, I didn't view Python as a functional programming language.

—Guido van Rossum, Python BDFL

FUNCTIONS AS FIRST-CLASS OBJECTS

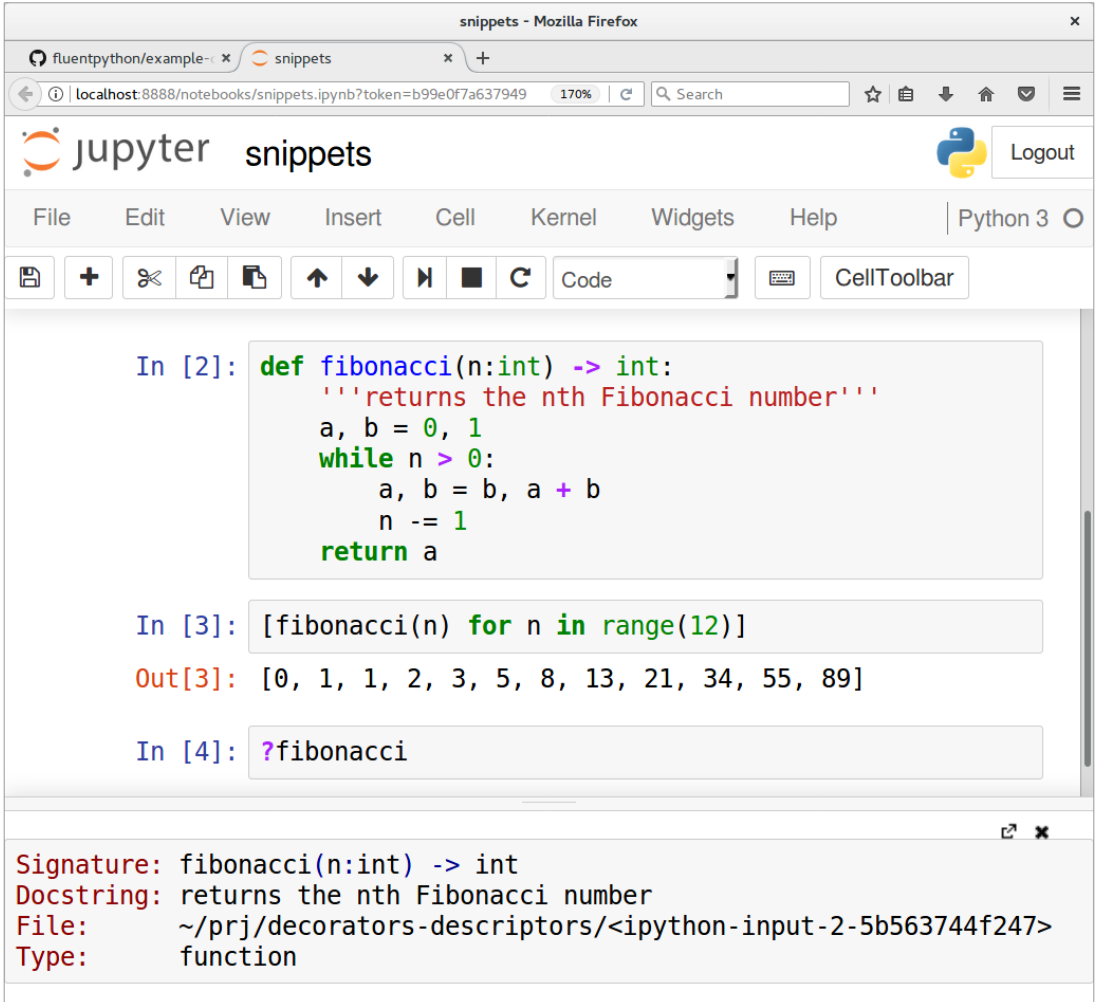
Python functions can be:

- Created at runtime
- Assigned to a variable or element in a data structure
- Passed as an argument to a function
- Returned as the result of a function

```
In [1]: fruit = ['banana', 'grapefruit', 'lime', 'pineapple']
        sorted(fruit, key=len)

Out[1]: ['lime', 'banana', 'pineapple', 'grapefruit']
```

FUNCTIONS HAVE ATTRIBUTES



FUNCTIONS HAVE ATTRIBUTES (2)

```
In [5]: fibonacci.__doc__
Out[5]: 'returns the nth Fibonacci number'

In [6]: fibonacci.__annotations__
Out[6]: {'n': int, 'return': int}

In [7]: fibonacci.__code__.co_varnames
Out[7]: ('n', 'a', 'b')

In [8]: from inspect import signature
        signature(fibonacci).parameters
Out[8]: mappingproxy({'n': <Parameter "n:int">})
```

ThoughtWorks®

DECORATORS 101

The basics

DECORATORS ARE SYNTACTIC SUGAR

These snippets have the same effect:

```
def square(n):
    return n * n

square = floatify(square)
```

```
@floatify
def square(n):
    return n * n
```

FUNCTION REPLACEMENT

Decorators may replace the decorated function with another function:

In [30]:
```
def deco(f):
    def inner():
        return 'inner result'
    return inner

@deco
def target():
    return 'original result'

target()
```

Out[30]:
```
'inner result'
```

In [31]:
```
target
```

Out[31]:
```
<function __main__.deco.<locals>.inner>
```

DECORATORS RUN AT IMPORT TIME

Import time == when a module is loaded

```
>>> import registration
running register(<function f1 at 0x…f28>)
>>>
```

DECORATORS RUN AT IMPORT TIME (2)

```
1  registry = []
2
3  def register(func):
4      print('running register(%s)' % func)
5      registry.append(func)
6      return func
7
8  @register
9  def f1():
10     print('running f1()')
11
12 if __name__ == '__main__':
13     print('running top level of module')
14     print('registry: ', registry)
15     f1()
```

```
>>> import registration
running register(<function f1 at 0x…f28>)
>>>
```

DECORATORS RUN AT IMPORT TIME (3)

```
1  registry = []
2
3  def register(func):
4      print('running register(%s)' % func)
5      registry.append(func)
6      return func
7
8  @register
9  def f1():
10     print('running f1()')
11
12 if __name__ == '__main__':
13     print('running top level of module')
14     print('registry: ', registry)
15     f1()
```

```
$ python3 registration.py
running register(<function f1 at 0x…7b8>)
running top level of module
registry:  [<function f1 at 0x…7b8>]
running f1()
```

ThoughtWorks®

EXERCISE 1: EXECUTION TIME

What happens when

SAMPLE EXERCISE

sample.py

```
1  print('<1>')
2
3  registry = []
4
5  def register(func):
6      print('<2>')
7      registry.append(func)
8      return func
9
10 @register
11 def f1():
12     print('<3>')
13
14 if __name__ == '__main__':
15     print('<4>')
16     f1()
```

Running this...

...will output this

```
$ python3 sample.py
<1>
<2>
<4>
<3>
```

EXERCISE 1: WRITE DOWN THE OUTPUT

main.py

```
1  from util import deco
2
3  print('<1>')
4
5  @deco
6  def first():
7      print('<2>')
8
9  @deco
10 def second():
11     third()
12     print('<3>')
13
14 def third():
15     print('<4>')
16
17 if __name__=='__main__':
18     first()
19     second()
20     print('<6>')
```

util.py

```
1  print('<A>')
2
3  def deco(f):
4      print('<B>')
5      def inner():
6          print('<C>')
7          f()
8      return inner
9
10 print('<D>')
```

```
$ python3 main.py
<?>
...
```

Write the expected output on paper. Don't use the computer.

ThoughtWorks®

REGISTRATION DECORATORS

The simplest kind

REGISTRATION DECORATORS

Goal:

Register decorated functions in a global application registry

Typical use case:

Register view functions in Web framework

SIMPLE EXAMPLE

A command-line utility that can be extended by adding decorated functions.

Example inspired by Armin Ronacher's ***click*** library for CLI:

http://click.pocoo.org/

```
$ ./kron.py
Usage:  ./kron.py d|m|t

$ ./kron.py t
20:24:02

$ ./kron.py d
Monday, May 01, 2017

$ ./kron.py m
      May 2017
Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

SIMPLE EXAMPLE (2)

The utility that can be extended by adding functions decorated with **@command**

```
#!/usr/bin/env python3

"""Time CLI didactic utility"""

from time import strftime, localtime
from calendar import prmonth

commands = {}

def command(f):
    initial = f.__name__[0]
    commands[initial] = f
    return f

@command
def time():
    print(strftime('%H:%M:%S'))

@command
def day():
    print(strftime('%A, %B %d, %Y'))

@command
def month():
    prmonth(*localtime()[:2])

def main(argv):
    if len(argv) > 1 and argv[1] in commands:
        commands[argv[1]]()
    else:
        options = '|'.join(sorted(commands))
        print(f'Usage:  {argv[0]} {options}')

if __name__ == '__main__':
    import sys
    main(sys.argv)
```

SIMPLE EXAMPLE (3)

The **@command** decorator puts the function in the **commands** dict, with its initial letter as key.

```
1  #!/usr/bin/env python3
2
3  """Time CLI didactic utility"""
4
5  from time import strftime, localtime
6  from calendar import prmonth
7
8  commands = {}
9
10 def command(f):
11     initial = f.__name__[0]
12     commands[initial] = f
13     return f
14
15 @command
16 def time():
17     print(strftime('%H:%M:%S'))
18
```

SIMPLE EXAMPLE (4)

`main` gets the function using the command-line argument as key, and calls it (line 29)

```
19 @command
20 def day():
21     print(strftime('%A, %B %d, %Y'))
22
23 @command
24 def month():
25     prmonth(*localtime()[:2])
26
27 def main(argv):
28     if len(argv) > 1 and argv[1] in commands:
29         commands[argv[1]]()
30     else:
31         options = '|'.join(sorted(commands))
32         print(f'Usage:  {argv[0]} {options}')
33
34 if __name__ == '__main__':
35     import sys
36     main(sys.argv)
```

SIMPLE EXERCISE

Add an **y** (year) command line option.
Hint: look for the **calendar.prcal** function

```
$ ./kron.py y
                                  2017

      January               February               March
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
                   1            1  2  3  4  5            1  2  3  4  5
 2  3  4  5  6  7  8   6  7  8  9 10 11 12   6  7  8  9 10 11 12
 9 10 11 12 13 14 15  13 14 15 16 17 18 19  13 14 15 16 17 18 19
16 17 18 19 20 21 22  20 21 22 23 24 25 26  20 21 22 23 24 25 26
23 24 25 26 27 28 29  27 28                 27 28 29 30 31
30 31

       April                  May                   June
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
                1  2   1  2  3  4  5  6  7            1  2  3  4
 3  4  5  6  7  8  9   8  9 10 11 12 13 14   5  6  7  8  9 10 11
10 11 12 13 14 15 16  15 16 17 18 19 20 21  12 13 14 15 16 17 18
17 18 19 20 21 22 23  22 23 24 25 26 27 28  19 20 21 22 23 24 25
```

SIMPLE EXERCISE SOLUTION

Code to add
a **y** (year) option

```
@command
def year():
    calendar.prcal(localtime()[0])
```

```
$ ./kron.py y
                                  2017

      January               February               March
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
                   1            1  2  3  4  5            1  2  3  4  5
 2  3  4  5  6  7  8   6  7  8  9 10 11 12   6  7  8  9 10 11 12
 9 10 11 12 13 14 15  13 14 15 16 17 18 19  13 14 15 16 17 18 19
16 17 18 19 20 21 22  20 21 22 23 24 25 26  20 21 22 23 24 25 26
23 24 25 26 27 28 29  27 28                 27 28 29 30 31
30 31

       April                  May                   June
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
                1  2   1  2  3  4  5  6  7            1  2  3  4
 3  4  5  6  7  8  9   8  9 10 11 12 13 14   5  6  7  8  9 10 11
10 11 12 13 14 15 16  15 16 17 18 19 20 21  12 13 14 15 16 17 18
17 18 19 20 21 22 23  22 23 24 25 26 27 28  19 20 21 22 23 24 25
```

ThoughtWorks®

VARIABLE SCOPE

Implicit vs. explicit variable declarations

LOCAL VS. GLOBAL VARIABLES

In a clean environment, define and call **f1**:

```
>>> def f1(a):
...     print(a)
...     print(b)
...
>>> f1(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f1
NameError: global name 'b' is not defined
```

To fix, create a global **b**, and call **f1** again:

```
>>> b = 6
>>> f1(3)
3
6
```

LOCAL VS. GLOBAL VARIABLES (2)

Function **f2** always fails, regardless of the environment:

```
>>> b = 6
>>> def f2(a):
...     print(a)
...     print(b)
...     b = 9
...
>>> f2(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f2
UnboundLocalError: local variable 'b' referenced before assignment
```


This assignment makes **b** a local variable.

THE global STATEMENT

```
>>> b = 6
>>> def f3(a):
...     global b
...     print(a)
...     print(b)
...     b = 9
...
>>> f3(3)
3
6
>>> b
9
```


With the **`global`** instruction, the compiler knows that **b** is global despite the assignment in the body of the function.

ThoughtWorks®

CLOSURES

Not the same as “anonymous functions”

THE RUNNING AVERAGE EXAMPLE

Imagine an **avg** function that computes the running average of a series of values.

```
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

Each call to **avg** adds a term and returns the updated average.

CLASS-BASED RUNNING AVERAGE

One way to do it, using a class:

```
class Averager():

    def __init__(self):
        self.series = []

    def __call__(self, new_value):
        self.series.append(new_value)
        total = sum(self.series)
        return total/len(self.series)
```

CLASS-BASED RUNNING AVERAGE (2)

One way to do it, using a class:

```
class Averager():

    def __init__(self):
        self.series = []

    def __call__(self, new_value):
        self.series.append(new_value)
        total = sum(self.series)
        return total/len(self.series)
```

Demo:

```
>>> avg = Averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

FUNCTION-BASED RUNNING AVERAGE

The functional way, using a higher-order function **`make_averager`**:

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total/len(series)

    return averager
```

FUNCTION-BASED RUNNING AVERAGE

The functional way, using a higher-order function **`make_averager`**:

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total/len(series)

    return averager
```

Demo:

```
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

FUNCTION-BASED RUNNING AVERAGE

The functional way, using a higher-order function **`make_averager`**:

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total/len(series)

    return averager
```

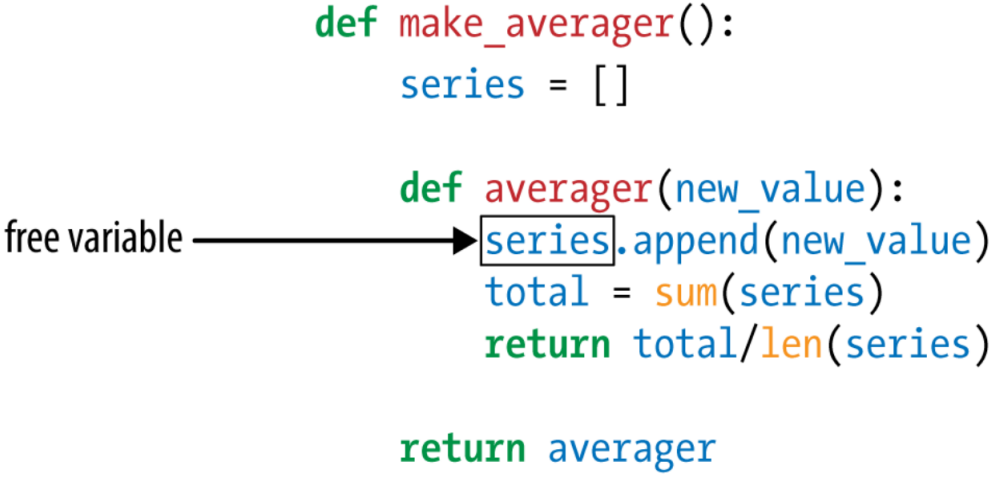
Demo:

```
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```


ThoughtWorks®
HOW DOES THIS WORK?
Closures FTW!

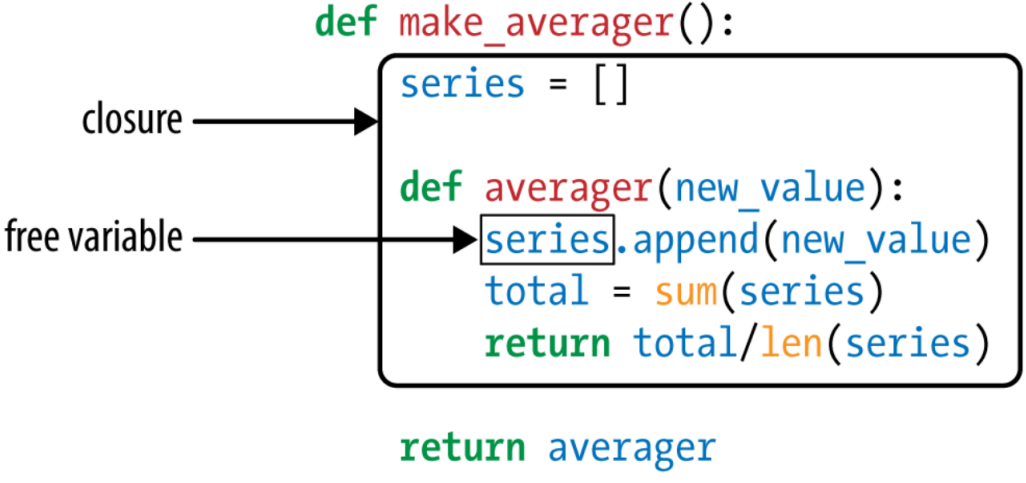
KEY CONCEPT: FREE VARIABLE

A ***free variable*** is a non-local variable referenced in a function body.



CLOSURES "ENCLOSE" FREE VARIABLES

The ***closure*** for **`averager`** encompasses the **`series`** free variable it needs to run.



FUNCTION-BASED RUNNING AVERAGE

The functional way, using a higher-order function **`make_averager`**:

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total/len(series)

    return averager
```

Demo:

```
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

A REAL CLOSURE UNDER THE MICROSCOPE

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total/len(series)

    return averager
```

```
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
>>> avg.__code__.co_varnames
('new_value', 'total')
>>> avg.__code__.co_freevars
('series',)
>>> avg.__closure__
(<cell at 0x107a44f78: list object at 0x107a91a48>,)
>>> avg.__closure__[0].cell_contents
[10, 11, 12]
```

BETTER RUNNING AVERAGE (BROKEN)

Keeping track of **`count`** and **`total`**, instead of the whole series.

However, this code fails.

Can you explain why?

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        count += 1
        total += new_value
        return total / count

    return averager
```

BETTER RUNNING AVERAGE (BROKEN)

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        count += 1
        total += new_value
        return total / count

    return averager
```

This is why:

```
>>> avg = make_averager()
>>> avg(10)
Traceback (most recent call last):
  ...
UnboundLocalError: local variable 'count' referenced before assignment
```

BETTER RUNNING AVERAGE (FIXED)

Use the **`nonlocal`** statement to declare variables that will be assigned in the function but are not local.

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        nonlocal count, total
        count += 1
        total += new_value
        return total / count

    return averager
```

BETTER RUNNING AVERAGE (FIXED)

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        nonlocal count, total
        count += 1
        total += new_value
        return total / count

    return averager
```

Fixed:

```
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

ThoughtWorks®

FUNCTIONAL DECORATORS

Altering behavior by function wrapping

VERY SIMPLE DECORATOR

```
1  """Simple decorator example"""
2
3  def floatify(f):
4      def floated(n):
5          result = f(n)
6          return float(result)
7      return floated
```

Replace decorated function **f** with **floated**, which:

- Calls **f(n)**
- Applies **float** to **result** and returns it

VERY SIMPLE DECORATOR DEMO

In [9]:
```
def square(n):
    return n * n

square(3)
```

Out[9]:
```
9
```

In [10]:
```
from decolib import floatify
```

In [11]:
```
@floatify
def square(n):
    return n * n

square(3)
```

Out[11]:
```
9.0
```

In [12]:
```
square
```

Out[12]:
```
<function decolib.floatify.<locals>.floated>
```

ISSUES TO BE ADDRESSED

- The replacement function *usually* honors the contract of the decorated function:
  - Accept same number/kinds of args
  - Return result of compatible type
- The replacement function should preserve metadata from the decorated function
  - Important for debugging and other metaprogramming purposes

PROPERLY WRAPPED DECORATOR

Use of **`@functools.wraps()`**:

```
1  from functools import wraps
2
3  def floatify(f):
4
5      @wraps(f)
6      def floated(n):
7          result = f(n)
8          return float(result)
9
10     return floated
```

In [1]:
```
from decolib2 import floatify

@floatify
def square(n):
    """returns n squared"""
    return n * n

square(3)
```

Out[1]:
```
9.0
```

In [2]:
```
square
```

Out[2]:
```
<function __main__.square>
```

In [3]:
```
help(square)
```

```
Help on function square in module __main__:

square(n)
    returns n squared
```

In [4]:
```
??square
```

```
Signature: square(n)
Source:
@floatify
def square(n):
    """returns n squared"""
    return n * n
File:      ~/prj/pycon/decorators-descriptors/<ipython-input-1-fd458bcaec78>
Type:      function
```

CLOCKING DECORATOR DEMO

@clock decorator displays:

- Elapsed time
- Arguments passed and results returned

```
$ python3 clockdeco_demo.py
**************************************** Calling snooze(.123)
[0.12818575s] snooze(0.123) -> None
**************************************** Calling factorial(6)
[0.00000191s] factorial(1) -> 1
[0.00002408s] factorial(2) -> 2
[0.00003886s] factorial(3) -> 6
[0.00005102s] factorial(4) -> 24
[0.00006604s] factorial(5) -> 120
[0.00008702s] factorial(6) -> 720
```

CLOCKING DECORATOR CODE

```
1  import time
2  from functools import wraps
3
4
5  def clock(func):
6
7      @wraps(func)
8      def clocked(*args):
9          t0 = time.time()
10         result = func(*args)
11         elapsed = time.time() - t0
12         name = func.__name__
13         arg_str = ', '.join(repr(arg) for arg in args)
14         print('[%0.8fs] %s(%s) -> %r' %
15               (elapsed, name, arg_str, result))
16         return result
17
18     return clocked
```

PARAMETRIZED CLOCKING DECORATOR

```
3  DEFAULT_FMT = '[{elapsed:0.8f}s] {name}({args}) -> {result}'
4
5  def clock(fmt=DEFAULT_FMT):
6      def decorate(func):
7          def clocked(*_args):
8              t0 = time.time()
9              _result = func(*_args)
10             elapsed = time.time() - t0
11             name = func.__name__
12             args = ', '.join(repr(arg) for arg in _args)
13             result = repr(_result)
14             print(fmt.format(**locals()))
15             return _result
16         return clocked
17     return decorate
```

CLASS-BASED CLOCKING DECORATOR

```
1  DEFAULT_FMT = '[{elapsed:0.8f}s] {name}({args}) -> {result}'
2
3  class Clocker:
4
5      def __init__(self, fmt=DEFAULT_FMT):
6          self.fmt = fmt
7
8      def __call__(self, func):
9          def clocked(*_args):
10             t0 = time.time()
11             _result = func(*_args)
12             elapsed = time.time() - t0
13             name = func.__name__
14             args = ', '.join(repr(arg) for arg in _args)
15             result = repr(_result)
16             print(self.fmt.format(**locals()))
17             return _result
18         return clocked
19
20 clock = Clocker()
```

DECORATE LIKE A PRO

- At a minimum, use **@functools.wraps**
- Use Graham Dumpleton's **wrapt**
  or
  Michele Simionato's **decorator** package

ThoughtWorks®

the black theme

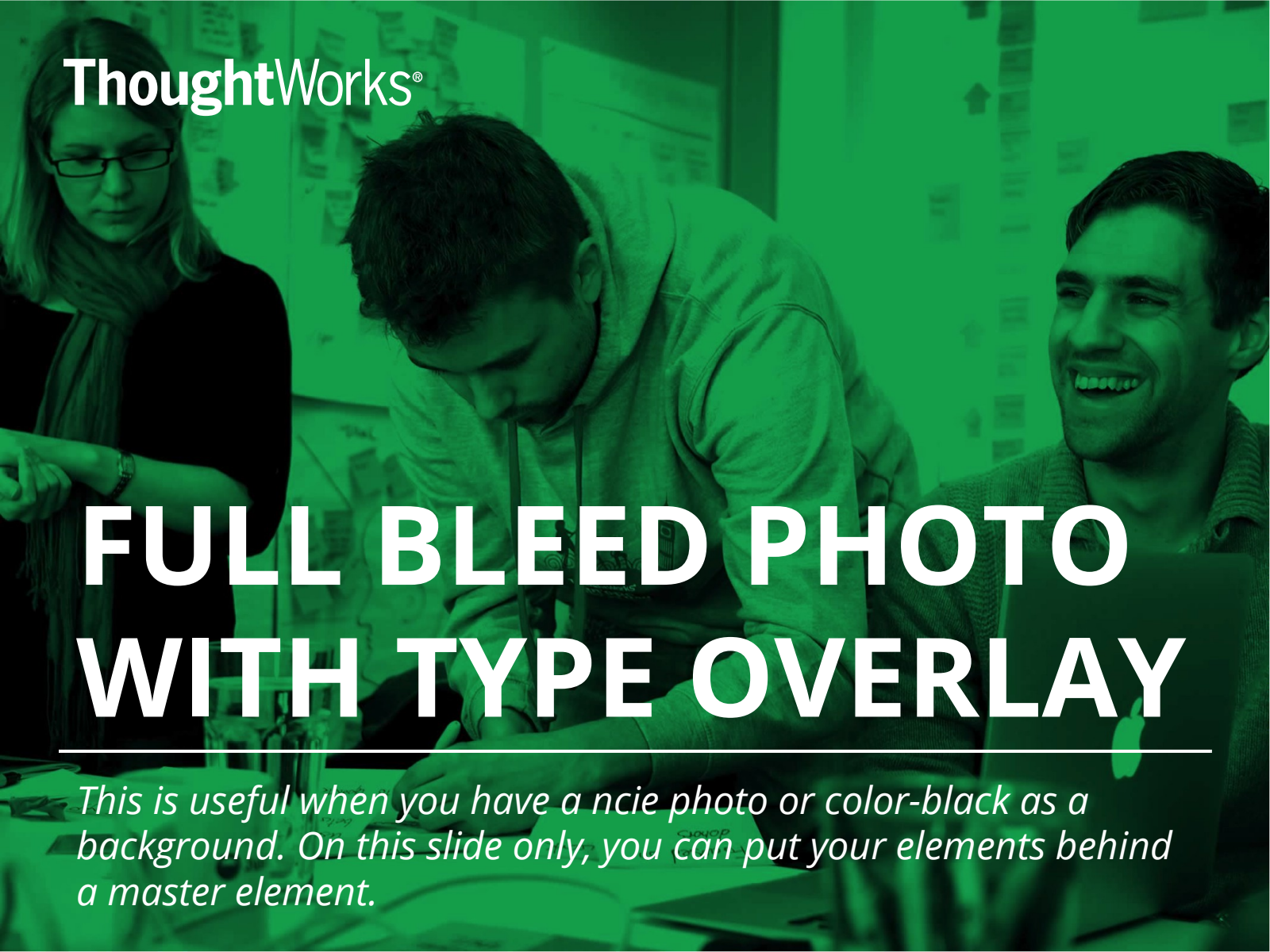
A NICE BIG TITLE

And then a subhead that can run long to add some background flavor or prosaic flourish.

ThoughtWorks®

ANOTHER WAY TO DO A BIG TITLE OR SECTION DIVIDER

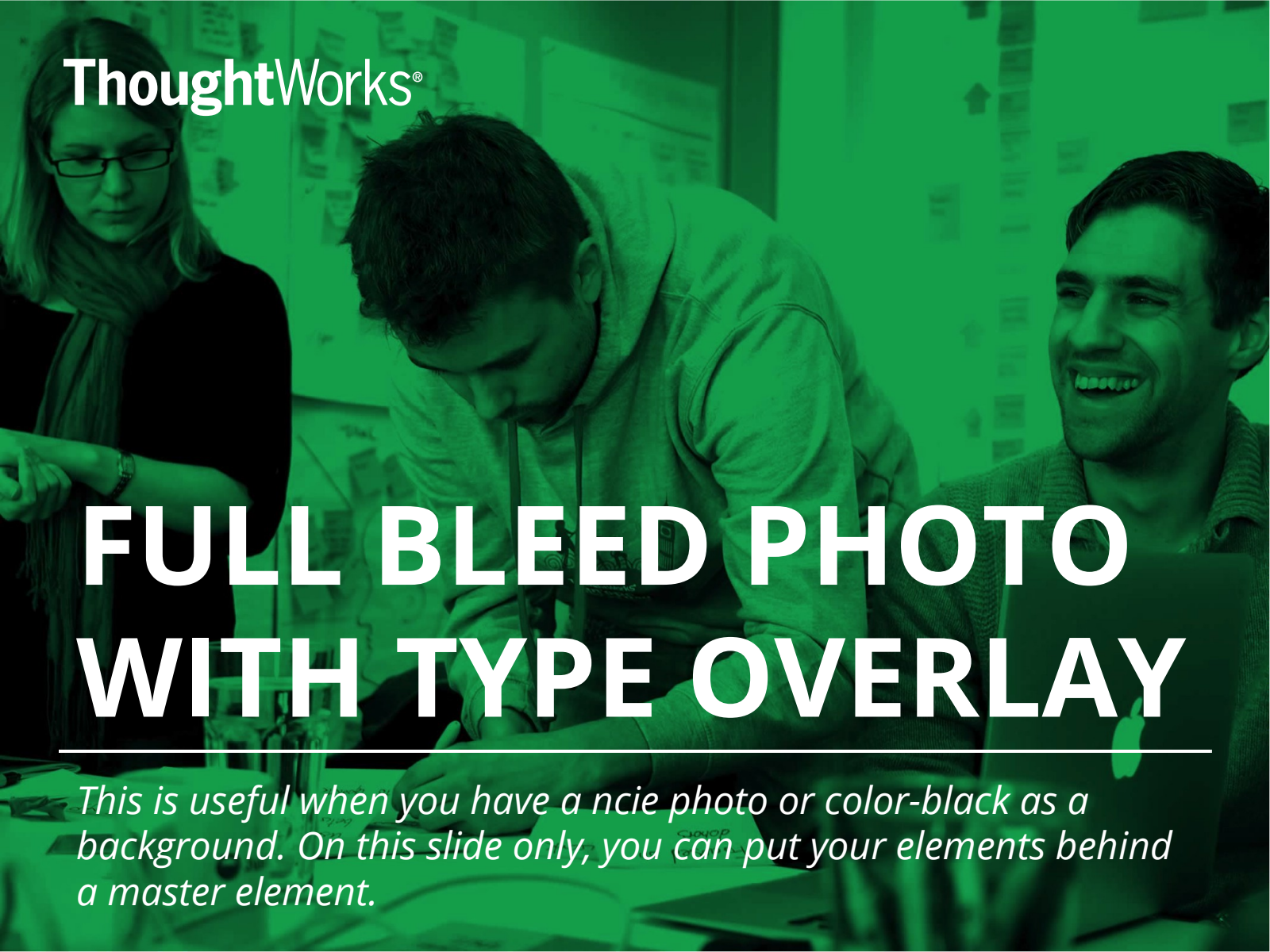
Add a subhead if you want.



ThoughtWorks®

FULL BLEED PHOTO WITH TYPE OVERLAY

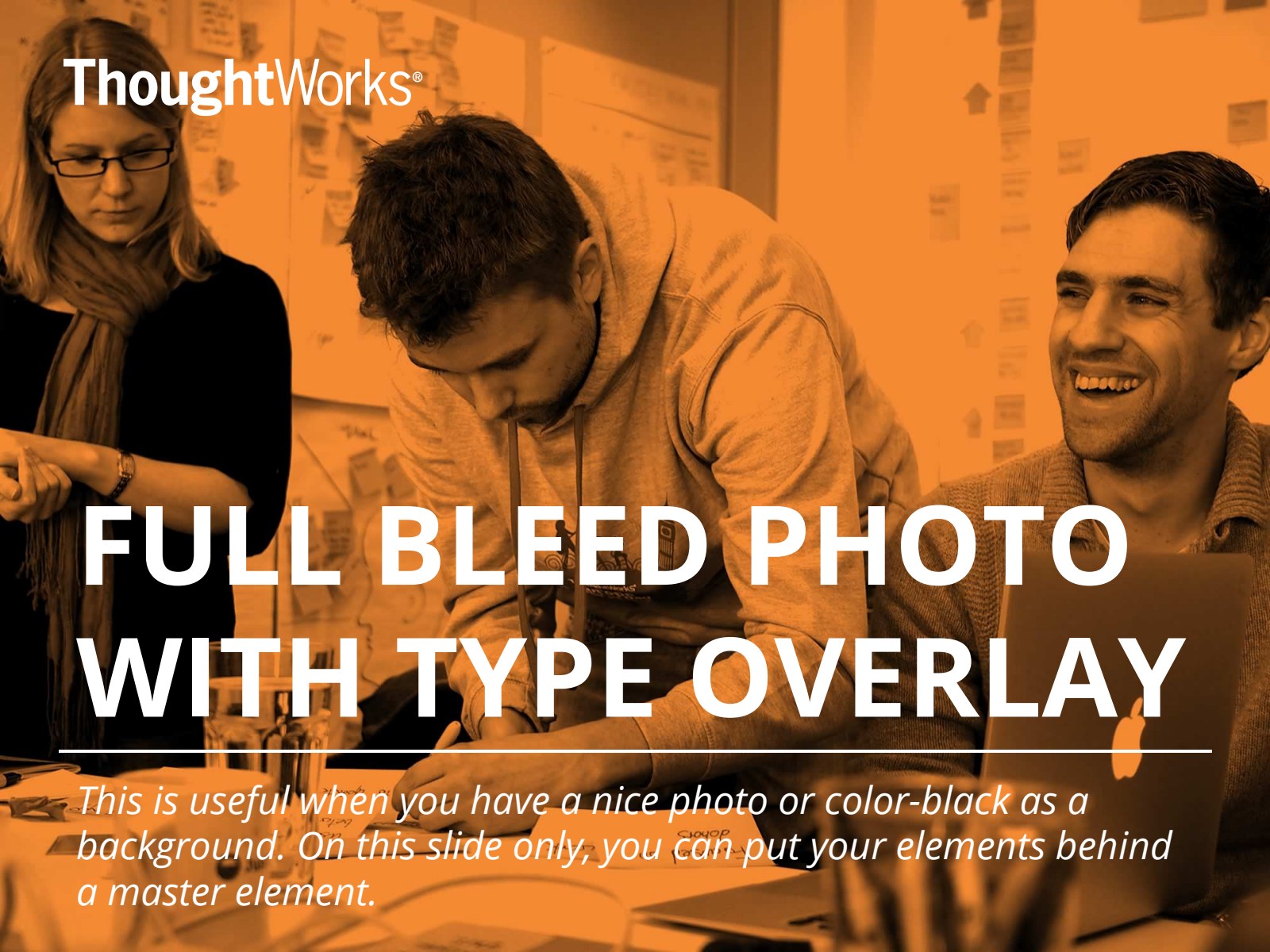
This is useful when you have a ncie photo or color-black as a background. On this slide only, you can put your elements behind a master element.



ThoughtWorks®

FULL BLEED PHOTO WITH TYPE OVERLAY

This is useful when you have a ncie photo or color-black as a background. On this slide only, you can put your elements behind a master element.



ThoughtWorks®

FULL BLEED PHOTO WITH TYPE OVERLAY

This is useful when you have a nice photo or color-black as a background. On this slide only, you can put your elements behind a master element.


ThoughtWorks®
UNTINTED PHOTO
Use with care.

YOUR BASIC, HARD-WORKING SLIDE

Make subheads 24

Body defaults to 18pt by default.

- Bullet One
  - Bullet Two
    - Bullet Three
    - Bullet Four
      - Bullet Five
        - Bullet 6

YOUR BASIC, HARD-WORKING SLIDE

Make subheads 24

Body defaults to 18pt by default.

- Bullet One
  - Bullet Two
    - Bullet Three
    - Bullet Four
      - Bullet Five
        - Bullet 6

50/50 CONTENT SPLIT

Make subheads 24

Body defaults to 18pt by default.

- Bullet One
  - Bullet Two
    - Bullet Three
    - Bullet Four
      - Bullet Five
        - Bullet 6

LONG FORM CONTENT

“Sometimes you need a quote and you’ll have to copy this block since there’s no way to predefine a quote style.”

Sometimes we use PowerPoint for page layout.

And in that case we need room for body copy. That’s what this slide is for.

Cras mattis consectetur purus sit amet fermentum. Duis mollis, est non commodo luctus, nisi erat porttitor ligula, eget lacinia odio sem nec elit. Morbi leo risus, porta ac consectetur ac, vestibulum at eros. Donec sed odio dui. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Praesent commodo cursus magna, vel scelerisque nisl consectetur et. Aenean lacinia bibendum nulla sed consectetur.

Sometimes you need a blank template.

Sometimes you want to say something simply and boldly on a dedicated slide. This template is called the "quote".

BLANK PAGE WITH A TITLE

COLOR PALETTE

Safe on Projectors

#0078bf 0,120,191	#00bccd 0,188,205	#7dced5 125,206,213	#00aa5b 0,170,91	#85b880 133,184,128	#bdbd32 189,189,50
#fff350 255,243,80	#fbe0ce 251,224,206	#f2ba97 242,186,151	#a17861 161,120,97	#808184 128,129,132	#eeeeee 238,238,238

Risky on Projectors

#ee5ba0 238,91,160	#702269 112,34,105	#b51b58 181,27,88	#ed312f 237,49,47	#f58a33 245,138,51	#5f3c25 95,60,37

BOXES, SHAPES AND TABLES

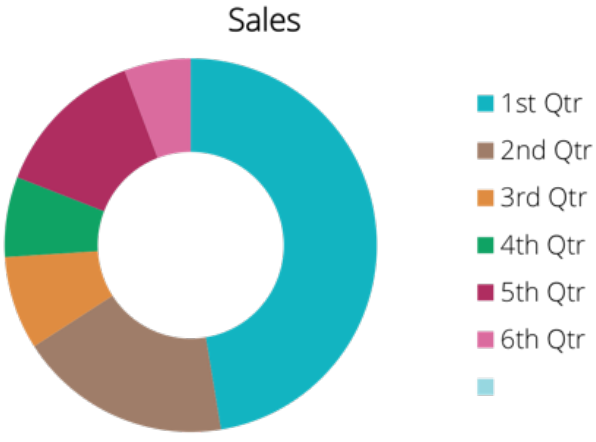
Buy default, a floating text box looks like this.

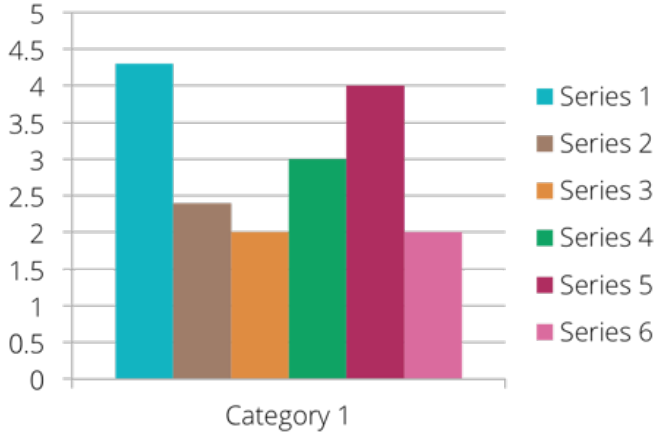
Shapes are tinted gray with lots of padding.

Works for any shape.

PowerPoint doesn't …	offer much …	in the way …
of custom …	table …	styles.
It prefers to use predefined	colors from …	The built in theme.

CHART SUGGESTIONS





CHECKLIST TEMPLATE

- ❏ First, get the proper font: http://opensans.com/
- ❏ Start with a clean template.
- ❏ When in doubt, copy and paste from something that already works.
- ❏ Be brave:
  - ❏ Learn to use master pages properly.
  - ❏ Edit! Brevity, not design, makes presentations great.

THANK YOU

For questions or suggestions:

Chad Currie

Smith & Robot

chad@smithandrobot.com

ThoughtWorks®