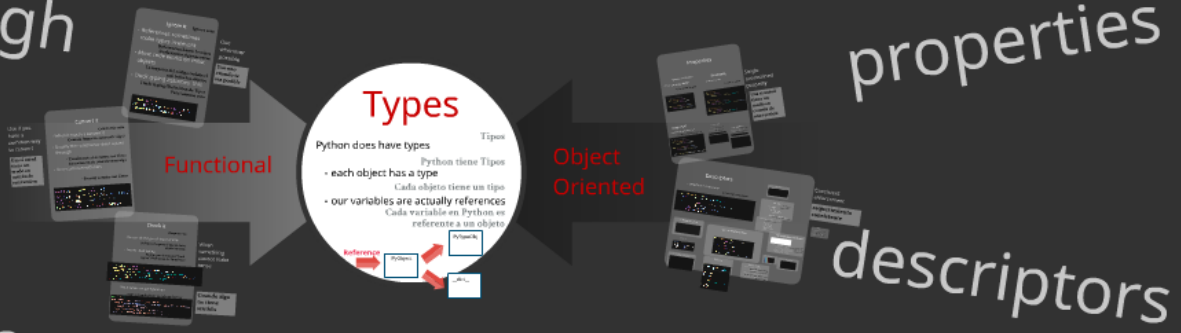
passthrough

typecast

isinstance



properties

descriptors

Python: As much typing as you want

Andy Fundinger
Senior Consultant
Risk Focus Inc

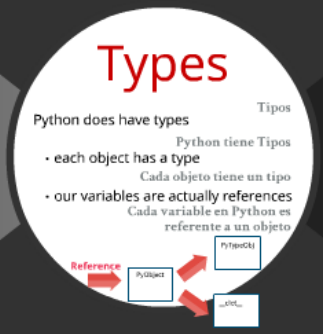
Andy.Fundinger@RiskFocus.com

passthrough

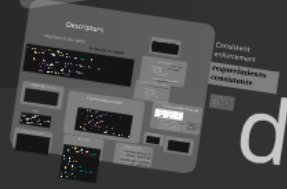
typecast

isinstance

Functional



Object Oriented



properties

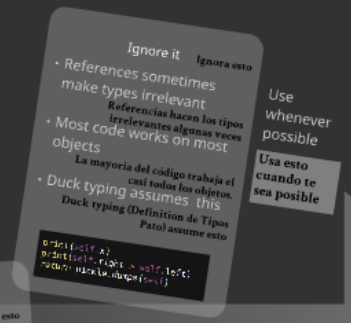
descriptors

Python: As much typing as you want

Andy Fundinger
Senior Consultant
Risk Focus Inc

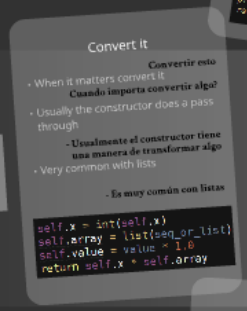
Andy.Fundinger@RiskFocus.com

gh



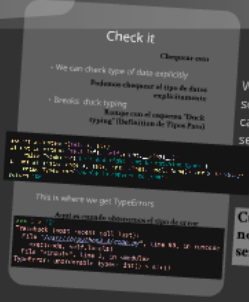
Use whenever possible

Usa esto cuando te sea posible



Use if you have a common way to convert

Usa si usted tiene un modo en común de conversion



When something cannot make sense

Cuando algo no tiene sentido

Functional

Types

Tipos

Python does have types

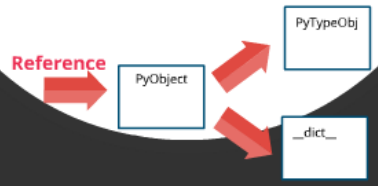
Python tiene Tipos

- each object has a type

Cada objeto tiene un tipo

- our variables are actually references

Cada variable en Python es referente a un objeto



Object Oriented

Single constrained property

Usa si usted tiene un modo en común de conversion

e

Python: As much typing as you want

Andy Fundinger
Senior Consultant
Risk Focus Inc

Andy.Fundinger@RiskFocus.com

Andy Fundinger
Senior Consultant
Risk Focus Inc

About Me

- New York based
  Basado en NYC
- Started Python at OSCON 2004 with Plone
  comenze a usar Python en OSCON 2014 con Plone
- Risk Platforms and Cloud Infrastructure projects
  Construyo Plataformas de Medicion de Riesgos en la Nube
- Python and cloud training
  enseno {Python y manejo de Nube

Andy.Fundinger@RiskFocus.com

About Me

- New York based

  Basado en NYC

- Started Python at OSCON 2004 with Plone

  comenze a usar Python en OSCON 2014 con Plone

- Risk Platforms and Cloud Infrastructure projects

  Construyo Plataformas de Medicion de Riesgos en la Nube

- Python and cloud training

  enseno {Python y manejo de Nube

Types

Tipos

Python does have types

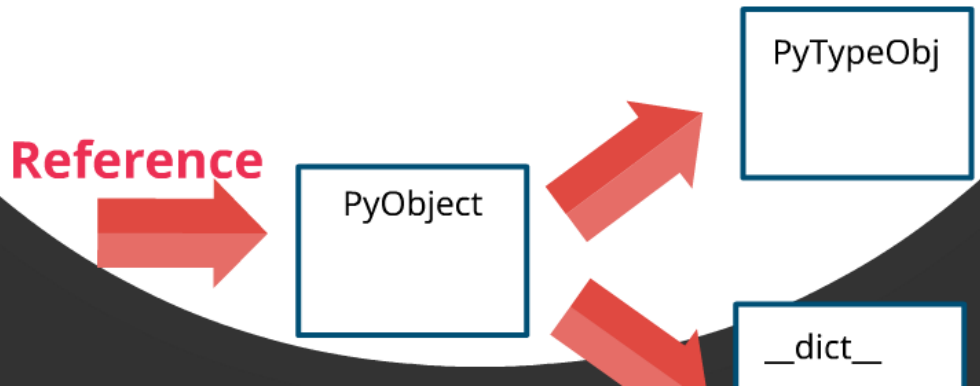
Python tiene Tipos

- each object has a type

Cada objeto tiene un tipo

- our variables are actually references

Cada variable en Python es referente a un objeto

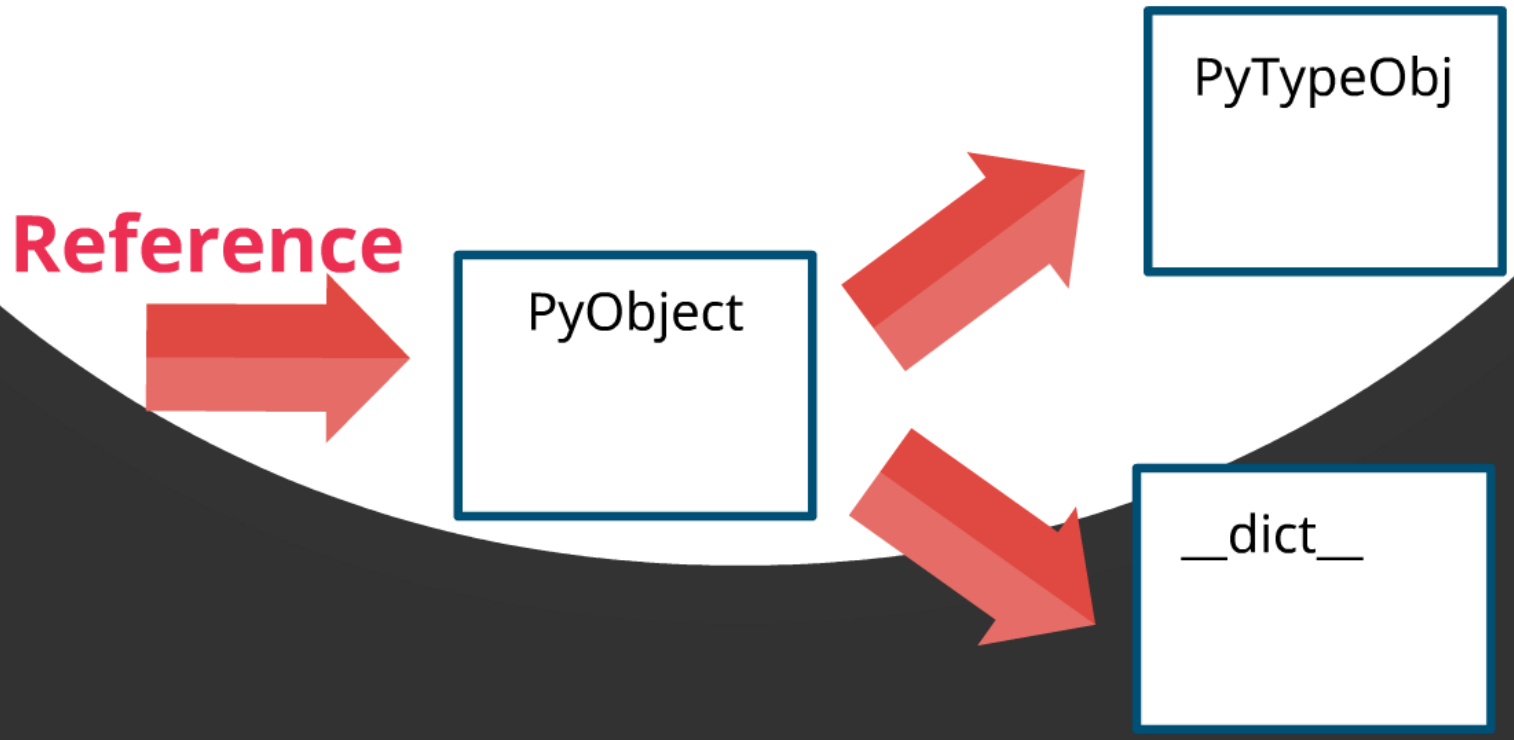


- each object has a type

Cada objeto tiene un tipo

- our variables are actually references

Cada variable en Python es referente a un objeto



hrough

ast

tance

Use if you have a common way to convert

Usa si usted tiene un modo en común de conversion

Convert it

Convertir esto

- When it matters convert it

Cuando importa convertir algo?

- Usually the constructor does a pass through

- Usualmente el constructor tiene una manera de transformar algo

- Very common with lists

- Es muy común con listas

```
self.x = int(self.x)
self.array = list(seq_or_list)
self.value = value * 1.0
return self.x * self.array
```

Ignore it

Ignora esto

- References sometimes make types irrelevant

Referencias hacen los tipos irrelevantes algunas veces

- Most code works on most objects

La mayoria del código trabaja el casi todos los objetos.

- Duck typing assumes this

Duck typing (Definition de Tipos Pato) assume esto

```
print(self.x)
print(self.right > self.left)
return pickle.dumps(self)
```

Use whenever possible

Usa esto cuando te sea posible

Check it

Chequear esto

- We can check type of data explicitly

Podemos chequear el tipo de datos explícitamente

- Breaks duck typing

Rompe con el esquema "Duck typing" (Definition de Tipos Pato)

This is where we get TypeErrors

Aqui es cuando obtenemos el tipo de error

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unorderable types: int() > str()
```

When something cannot make sense

Cuando algo no tiene sentido

Functional

Types

Python does have types

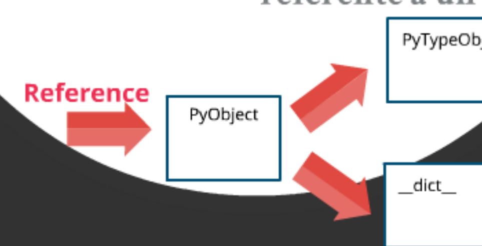
Python tiene

- each object has a type

Cada objeto tiene u

- our variables are actually refer

Cada variable en Pyt referente a un



Python: As mu

Ignore it — Ignora esto

- References sometimes make types irrelevant

  Referencias hacen los tipos irrelevantes algunas veces
- Most code works on most objects

  La mayoria del código trabaja el casi todos los objetos.
- Duck typing assumes this

  Duck typing (Definition de Tipos Pato) assume esto

```
print(self.x)
print(self.right > self.left)
return pickle.dumps(self)
```

Use whenever possible

Usa esto cuando te sea posible

objects

La mayoria del código trabaja el casi todos los objetos.

- Duck typing assumes this

Duck typing (Definition de Tipos Pato) assume esto

```
print(self.x)
print(self.right > self.left)
return pickle.dumps(self)
```

se if you
ve a
mmon way
convert

sa si usted
ene un
odo en
omún de
onversion

```
return pickle.
```

Convert it

Convertir esto

- When it matters convert it

Cuando importa convertir algo?

- Usually the constructor does a pass through

- Usualmente el constructor tiene una manera de transformar algo

- Very common with lists

- Es muy común con listas

```
self.x = int(self.x)
self.array = list(seq_or_list)
self.value = value * 1.0
return self.x * self.array
```

una manera de transformar algo

- Very common with lists

- Es muy común con listas

```
self.x = int(self.x)
self.array = list(seq_or_list)
self.value = value * 1.0
return self.x * self.array
```

Check it

Chequear esto

- We can check type of data explicitly

Podemos chequear el tipo de datos explícitamente

- Breaks duck typing

Rompe con el esquema "Duck typing" (Definition de Tipos Pato)

```
assert isinstance(self.x, str)
if not isinstance(self.left, self.right.__class__):
    raise TypeError('Left and right must be matching types')
if not all(isinstance(x, (str, int, float, bool, None)) for x in vals):
    raise TypeError('Unable to convert to json')
return 'OK'
```

This is where we get TypeErrors

Aquí es cuando obtenemos el tipo de error

```
>>> 1 > '2'
Traceback (most recent call last):
  File "/usr/lib/python3.4/code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
TypeError: unorderable types: int() > str()
```

When something cannot make sense

Cuando alg no tiene sentido

- We can check type of data explicitly

Podemos chequear el tipo de datos explícitamente

- Breaks duck typing

Rompe con el esquema "Duck typing" (Definition de Tipos Pato)

```
assert isinstance(self.x, str)
if not isinstance(self.left, self.right.__class__):
    raise TypeError('Left and right must be matching types')
if not all(isinstance(x, (str, int, float, bool, None)) for x in vals):
    raise TypeError('Unable to convert to json')
return 'OK'
```

This is where we get TypeErrors

Aquí es cuando obtenemos el tipo de error

```
>>> 1 > '2'
Traceback (most recent call last):
  File "/usr/lib/python3.4/code.py", line 90, in runcode
    exec(code, self.locals)
```

```
if not isinstance(self.left, self.right.__class__):
    raise TypeError('Left and right must be matching types')
if not all(isinstance(x, (str, int, float, bool, None)) for x in vals):
    raise TypeError('Unable to convert to json')
return 'OK'
```

This is where we get TypeErrors

Aquí es cuando obtenemos el tipo de error

```
>>> 1 > '2'
Traceback (most recent call last):
  File "/usr/lib/python3.4/code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
TypeError: unorderable types: int() > str()
```

```
self.x = int(self.x)
self.array = list(seq_or_list)
self.value = value * 1.0
return self.x * self.array
```

Check it

Chequear esto

- We can check type of data explicitly

Podemos chequear el tipo de datos explícitamente

- Breaks duck typing

Rompe con el esquema "Duck typing" (Definition de Tipos Pato)

```
assert isinstance(self.x, str)
if not isinstance(self.left, self.right.__class__):
    raise TypeError('Left and right must be matching types')
if not all(isinstance(x, (str, int, float, bool, None)) for x in vals):
    raise TypeError('Unable to convert to json')
return 'OK'
```

This is where we get TypeErrors

Aquí es cuando obtenemos el tipo de error

```
>>> 1 > '2'
Traceback (most recent call last):
  File "/usr/lib/python3.4/code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
TypeError: unorderable types: int() > str()
```

When something cannot make sense

Cuando algo no tiene sentido

asstrough

pecast

sinstance

Functional

Types

Tipos

Python does have types

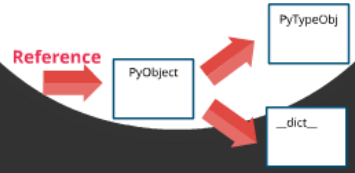
Python tiene Tipos

- each object has a type

Cada objeto tiene un tipo

- our variables are actually references

Cada variable en Python es referente a un objeto



Obje
Orien

Python: As much t

as you want

Types

Tipos

does have types

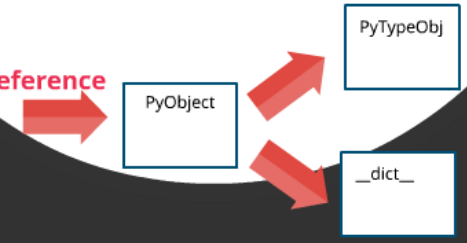
Python tiene Tipos

h object has a type

Cada objeto tiene un tipo

variables are actually references

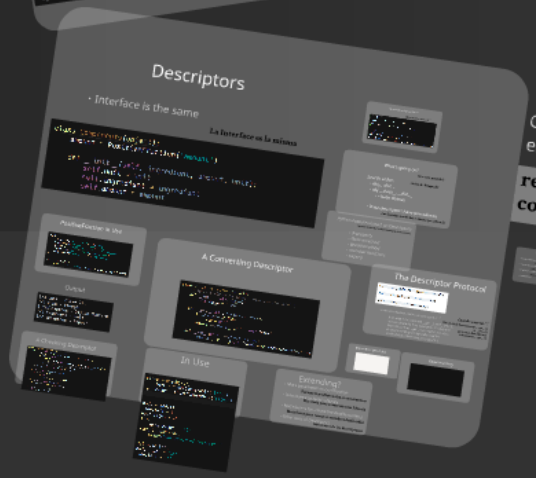
Cada variable en Python es referente a un objeto



Object Oriented

Single constrained property

Usa si usted tiene un modo en común de conversion



Consistent enforcement

requerimiento consistente

pro

desc

s much typing

Properties

Getters and Setters

- Cross Language Option

Opcion de Multilenguaje

```
# Getter/Setter
def get_age(self):
    return self.__age

def set_age(self, age):
    self.__age = int(age)
```

@property

- Used as a decorator

Usado como un decorador

```
@property
def name(self):
    return self.__dict__['name']

@name.setter
def name(self, value):
    self.__dict__['name'] = str(value)

@name.deleter
def name(self):
    del self.__dict__['name']
```

property()

- Explicit use of getter and setter

uso explicito como un getter y setter

```
# Getter/Setter
def get_age(self):
    return self.__age

def set_age(self, age):
    self.__age = int(age)

age = property(get_age, set_age)
```

Closures & property()

- Create closures

crear closures (clausuras)

- Pass to property() with *args

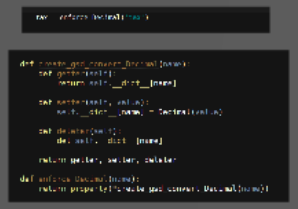
Pasar argumentos a property() con *args

Further simplification

Simplicacion adicional

- moving the property() call into the function

Mover la llamada property() en la funcion



Single constrained property

Usa si usted tiene un modo en común de conversion

Getters and Setters

- Cross Language Option

Opcion de Multilenguaje

```
# Getter/Setter
def get_age(self):
    return self.__age

def set_age(self, age):
    self.__age = int(age)
```

- Used as

```
@property
def name
    retu

@name.se
def name
    self

@name.de
def name
```

@property

- Used as a decorator

Usado como un decorador

```
@property
def name(self):
    return self.__dict__['name']

@name.setter
def name(self, value):
    self.__dict__['name'] = str(value)

@name.deleter
def name(self):
    del self.__dict__['name']
```

Single constrained property

Usa si usted tiene un modo en común de

- Used as a decorator

Usado como un decorador

```
@property
def name(self):
    return self.__dict__['name']

@name.setter
def name(self, value):
    self.__dict__['name'] = str(value)

@name.deleter
```

```
@property
def name(self):
    return self.__dict__['name']

@name.setter
def name(self, value):
    self.__dict__['name'] = str(value)

@name.deleter
def name(self):
    del self.__dict__['name']
```

```
        return self.__dict__['name']

    @name.setter
    def name(self, value):
        self.__dict__['name'] = str(value)

    @name.deleter
    def name(self):
        del self.__dict__['name']
```

@property

- Used as a decorator

Usado como un decorador

```
@property
def name(self):
    return self.__dict__['name']

@name.setter
def name(self, value):
    self.__dict__['name'] = str(value)

@name.deleter
def name(self):
    del self.__dict__['name']
```

Single constrained property

Usa si usted tiene un modo en común de conversion

property()

- Explicit use of getter and setter

uso explicito como un getter y setter

```
# Getter/Setter
def get_age(self):
    return self.__age

def set_age(self, age):
    self.__age = int(age)

age = property(get_age, set_age)
```

Closures & property()

- Create closures

crear closures (clausuras)

- Pass to property() with *args

Pasar argumentos a property() con *args

```
price = property(*create_gsd_convert_Decimal('price'))
```

```
def create_gsd_convert_Decimal(name):
    def getter(self):
        return self.__dict__[name]
```

- Pass to property() with *args

Pasar argumentos a property() con *args

```
price = property(*create_gsd_convert_Decimal('price'))
```

```
def create_gsd_convert_Decimal(name):
    def getter(self):
        return self.__dict__[name]

    def setter(self, value):
        self.__dict__[name] = Decimal(value)

    def deleter(self):
        del self.__dict__[name]

    return getter, setter, deleter
```

Further simplification

Simplicacion adicional

- moving the property() call into the function

  Mover la llamada property() en la funcion

```
tax = enforce_Decimal('tax')
```

```
def create_gsd_convert_Decimal(name):
    def getter(self):
        return self.__dict__[name]
```

```
    tax = enforce_Decimal('tax')
```

```
def create_gsd_convert_Decimal(name):
    def getter(self):
        return self.__dict__[name]

    def setter(self, value):
        self.__dict__[name] = Decimal(value)

    def deleter(self):
        del self.__dict__[name]

    return getter, setter, deleter

def enforce_Decimal(name):
    return property(*create_gsd_convert_Decimal(name))
```

Properties

Getters and Setters

- Cross Language Option

Opcion de Multilenguaje

```
# Getter/Setter
def get_age(self):
    return self.__age

def set_age(self, age):
    self.__age = int(age)
```

@property

- Used as a decorator

Usado como un decorador

```
@property
def name(self):
    return self.__dict__['name']

@name.setter
def name(self, value):
    self.__dict__['name'] = str(value)

@name.deleter
def name(self):
    del self.__dict__['name']
```

property()

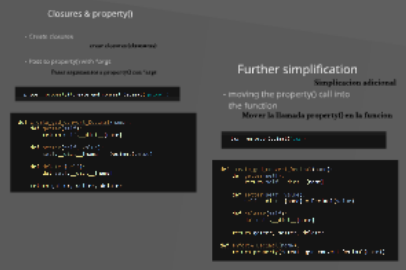
- Explicit use of getter and setter

uso explicito como un getter y setter

```
# Getter/Setter
def get_age(self):
    return self.__age

def set_age(self, age):
    self.__age = int(age)

age = property(get_age, set_age)
```



Single constrained property

Usa si usted tiene un modo en común de conversion

ct

nted

Descriptors

- Interface is the sa

Descriptors

- Interface is the same

La Interface es la misma

```
class Components(object):
    amount = PositiveFraction('amount')

    def __init__(self, ingredient, amount, unit):
        self.unit = unit
        self.ingredient = ingredient
        self.amount = amount
```

Something odd about the "."

Algo particular acerca de "."

What's going on?

Que esta pasando?

Search order: **Orden de Busqueda:**

- obj.__dict__
- obj.__class__.__dict__
  - + base classes

- Data descriptors have precedence

Los descriptores de datos tienen precedencia

Python Features based on Descriptors

Caracteristicas de Python basadas en descriptores

- @property
- @classmethod
- @staticmethod
- member functions
- super()

PositiveFraction in Use

```
recipe = [
    Components('Olive Oil', .75, 'cups'),
    Components('Vinegar', '.25', 'cups'),
    Components('Dijion Mustard', 1, 'tablespoons'),
    Components('Salt', '1/4', 'tomspoons'),
    Components('Pepper', 1 / 2, 'teaspoons')
]

for c in recipe:
    print('%s %s - %s'%(c.amount,c.unit, c.ingradient))
```

Output

```
3/4 cups - Olive Oil
1/4 cups - Vinegar
1 tablespoons - Dijion Mustard
1/4 teaspoons - Salt
1/2 teaspoons - Pepper
```

A Converting Descriptor

```
class PositiveFraction(object):
    '''Force field value to be a positive fraction'''
    def __init__(self, fname):
        self.fname = fname

    def __get__(self, instance, owner):
        if instance is None:
            return self
        return instance.__dict__[self.fname]

    def __set__(self, instance, value):
        instance.__dict__[self.fname] = max(Fraction(value), 0)
```

The Descriptor Protocol

```
descr.__get__(self, obj, type=None) --> value

descr.__set__(self, obj, value) --> None

descr.__delete__(self, obj) --> None
```

- Controls resolution when accessing with "."

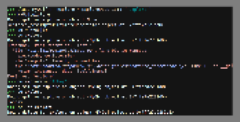
If an object defines both __get__() and __set__(), it is considered a data descriptor. Descriptors that only define __get__() are called non-data descriptors (they are typically used for methods but other uses are possible).

Cuando se accede "."

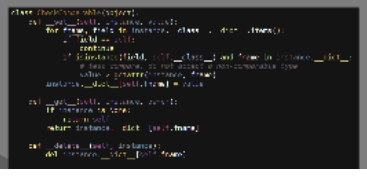
Para definir datos usamos __get__() y __set__() para otros descriptores solo usamos __get__()

Exploratory Descriptor

Experimenting



A Checking Descriptor



In Use

```
class Node(object):
    left = CheckComparable('left')
    right = CheckComparable('right')
```

Extending?

- More parameters in construction

Parametros adicionales en construccion

- Inheritance to build a library

Herencia para construir una libreria

- Metaclasses to remove the double naming

Metaclases para remover nombres duplicados

- Other uses of Descriptors

Otros usos de los descriptores

Something odd about the "."

Algo particular acerca de "."

```
>>> b = Booking()
>>> b.__dict__
{'name': '', '_Booking__age': None, 'room': None}
>>> b.floor = 3
>>> b.__dict__
{'_Booking__age': None, 'name': '', 'room': None, 'floor': 3}
>>> b.floor
3
>>> b.age = 42
>>> b.__dict__
{'_Booking__age': 42, 'name': '', 'room': None, 'floor': 3}
>>> Booking.get_age
<function Booking.get_age at 0x7f047ccf6bf8>
>>> b.get_age
<bound method Booking.get_age of <enforce_properties.Booking object at 0x7f047ccefcc0>>
```

What's going on?

Que esta pasando?

Search order:

Orden de Busqueda:

- obj.__dict__
- obj.__class__.__dict__
  - + base classes

- Data descriptors have precedence

Los descriptores de datos tienen precedencia

Python Features based on Descriptors

Caracteristicas de Python basadas en descriptores

Data descriptors have pr

Los descriptores de datos

Python Features based on Descriptors

Caracteristicas de Python basadas en descriptores

- @property
- @classmethod
- @staticmethod
- member functions
- super()

The

The Descriptor Protocol

```
descr.__get__(self, obj, type=None) --> value

descr.__set__(self, obj, value) --> None

descr.__delete__(self, obj) --> None
```

- Controls resolution when accessing with "."

If an object defines both __get__() and __set__(), it is considered a data descriptor. Descriptors that only define __get__() are called non-data descriptors (they are typically used for methods but other uses are possible).

Cuando se accede "."

Para definir datos usamos __get__() y __set__() para otros descriptores solo usamos __get__()

Exploratory Descriptor

```
class ExploreDescriptor(object):
    '''Exploratory Descriptor'''
    def __init__(self, fname):
        self.fname = fname

    def __get__(self, instance, owner):
        print('Doing get for %s on object %s'%(self.fname, instance))
        if instance is None:
            return self
        return instance.__dict__[self.fname]

    def __set__(self, instance, value):
        print('Doing set for %s on object %s'%(self.fname, instance))
        instance.__dict__[self.fname] = value

    def __delete__(self, instance):
        print('Doing delete for %s on object %s'%(self.fname, instance))
        del instance.__dict__[self.fname]
```

Experimenting

```
>>> class MyObj():  explore = ExploreDescriptor('explore')
>>> MyObj.explore
Doing get for explore on object None
<enforce_descriptor.ExploreDescriptor object at 0x7ffa7a221ba8>
>>> oo = MyObj()
>>> oo.explore
Doing get for explore on object <MyObj object at 0x7ffa7a2281d0>
Traceback (most recent call last):
  File "/usr/lib/python3.4/code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
  File "/home/andriod/Dropbox/Risk Focus/Typeless/enforce_descriptor.py", line 13, in __get
    return instance.__dict__[self.fname]
KeyError: 'explore'
>>> oo.explore = 'Abbey'
Doing set for explore on object <MyObj object at 0x7ffa7a2281d0>
>>> oo.explore
Doing get for explore on object <MyObj object at 0x7ffa7a2281d0>
'Abbey'
>>> del oo.explore
Doing delete for explore on object <MyObj object at 0x7ffa7a2281d0>
```

- me
- su

A Converting Descriptor

```
class PositiveFraction(object):
    '''Force field value to be a positive fraction'''
    def __init__(self, fname):
        self.fname = fname

    def __get__(self, instance, owner):
        if instance is None:
            return self
        return instance.__dict__[self.fname]

    def __set__(self, instance, value):
        instance.__dict__[self.fname] = max(Fraction(value), 0)
```

In Use

Extending

- More parameters in cons

Descriptors

- Interface is the same

La Interface es la misma

```
class Components(object):
    amount = PositiveFraction('amount')

    def __init__(self, ingredient, amount, unit):
        self.unit = unit
        self.ingredient = ingredient
        self.amount = amount
```

PositiveFraction in Use

PositiveFraction in Use

```
recipe = [
    Components('Olive Oil', .75, 'cups'),
    Components('Vinegar', '.25', 'cups'),
    Components('Dijion Mustard', 1, 'tablespoons'),
    Components('Salt', '1/4', 'teaspoons'),
    Components('Pepper', 1 / 2, 'teaspoons')
]

for c in recipe:
    print('%s %s - %s'%(c.amount,c.unit, c.ingredient))
```

Output

```
3/4 cups - Olive Oil
1/4 cups - Vinegar
1 tablespoons - Dijion Mustard
1/4 teaspoons - Salt
1/2 teaspoons - Pepper
```

A Checking Descriptor

```
class CheckComparable(object):
    def __set__(self, instance, value):
        for fname, field in instance.__class__.__dict__.items():
            if field == self:
                continue
            if isinstance(field, self.__class__) and fname in instance.__dict__:
                # test compare, do not accept a non-comparable type
                value > getattr(instance, fname)
        instance.__dict__[self.fname] = value

    def __get__(self, instance, owner):
        if instance is None:
            return self
        return instance.__dict__[self.fname]

    def __delete__(self, instance):
        del instance.__dict__[self.fname]

    def __init__(self, fname):
        self.fname = fname
```

In Use

```
class Node(object):
    left = CheckComparable('left')
    right = CheckComparable('right')


node = Node()
node.left = 5
node.right = 15
```

```
    right = CheckComparable('right')


node = Node()
node.left = 5
node.right = 15
try:
    node.right = '15'
except TypeError:
    print('Got Expected TypeError')

del node.left
node.right = '15'
node.left = '5'
```

Extending?

- More parameters in construction

  Parametros adicionales en constuccion

- Inheritance to build a library

  Herencia para construir una libreria

- Metaclasses to remove the double naming

  Metaclases para remover nombres duplicados

- Other uses of Descriptors

  Otros usos de los descriptores

• Explicit use of ge...

uso explicito como un ge...

```
# Getter/Setter
def get_age(self):
    return self.__age

def set_age(self, age):
    self.__age = int(age)

age = property(get_age, set_age)
```

Descriptors

- Interface is the same

La Interface es la misma

```
class Components(object):
    amount = PositiveFraction('amount')

    def __init__(self, ingredient, amount, unit):
        self.unit = unit
        self.ingredient = ingredient
        self.amount = amount
```

Something just about the "."

Algo particular acerca de "."

What's going on?

Que esta pasando?

Search order:

Orden de Busqueda:

- obj.__dict__
- obj.__class__.__dict__
  - + base classes

- Data descriptors have precedence

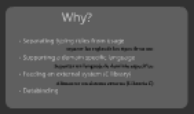
Los descriptores de datos tienen precedencia

Python Features based on Descriptors

Caracteristicas de Python basadas en descriptores

- @property
- @classmethod
- @staticmethod
- member functions
- super()

Why?



PositiveFraction in Use

Output

```
3/4 cups - Olive Oil
1/4 cups - Vinegar
1 tablespoons - Dijion Mustard
1/4 teaspoons - Salt
1/2 teaspoons - Pepper
```

A Converting Descriptor

```
class PositiveFraction(object):
    '''Force field value to be a positive fraction'''
    def __init__(self, fname):
        self.fname = fname

    def __get__(self, instance, owner):
        if instance is None:
            return self
        return instance.__dict__[self.fname]

    def __set__(self, instance, value):
        instance.__dict__[self.fname] = max(Fraction(value), 0)
```

The Descriptor Protocol

```
desc.__get__(self, obj, type=None) --> value

desc.__set__(self, obj, value) --> None

desc.__delete__(self, obj) --> None
```

- Controls resolution when accessing with "."

Cuando se accede "."

Para definir datos usamos __get__() y __set__()

para otros descriptores solo usamos __get__()

If an object defines both __get__() and __set__(), it is considered a data descriptor. Descriptors that only define __get__() are called non-data descriptors (they are typically used for methods but other uses are possible).

Exploratory Descriptor

Experimenting

A Checking Descriptor

In Use

```
class Node(object):
    left = CheckComparable('left')
    right = CheckComparable('right')


node = Node()
node.left = 5
node.right = 15
try:
    node.right = '15'
except TypeError:
    print('Got Expected TypeError')

del node.left
node.right = '15'
node.left = '5'
```

Extending?

- More parameters in construction

Parametros adicionales en constuccion

- Inheritance to build a library

Herencia para construir una libreria

- Metaclasses to remove the double naming

Metaclases para remover nombres duplicados

- Other uses of Descriptors

Otros usos de los descriptores

Consistent enforcement

requerimiento consistente

C

Why?

- Separating typing rules from usage

  separar las reglas de los tipos de su uso

- Supporting a domain specific language

  Soportar un lenguaje de dominio especifico

- Feeding an external system (C library)

  Alimentar un sistema externo (Libreria C)

- Databinding

Thank You!

Slides: http://bit.ly/217iyAl

Example Code:
https://github.com/riskfocus/Typeless

Muchas Gracias