# Domain Specific Languages in Python.
# Why and How.

## Doing (delightfully) weird things to Python

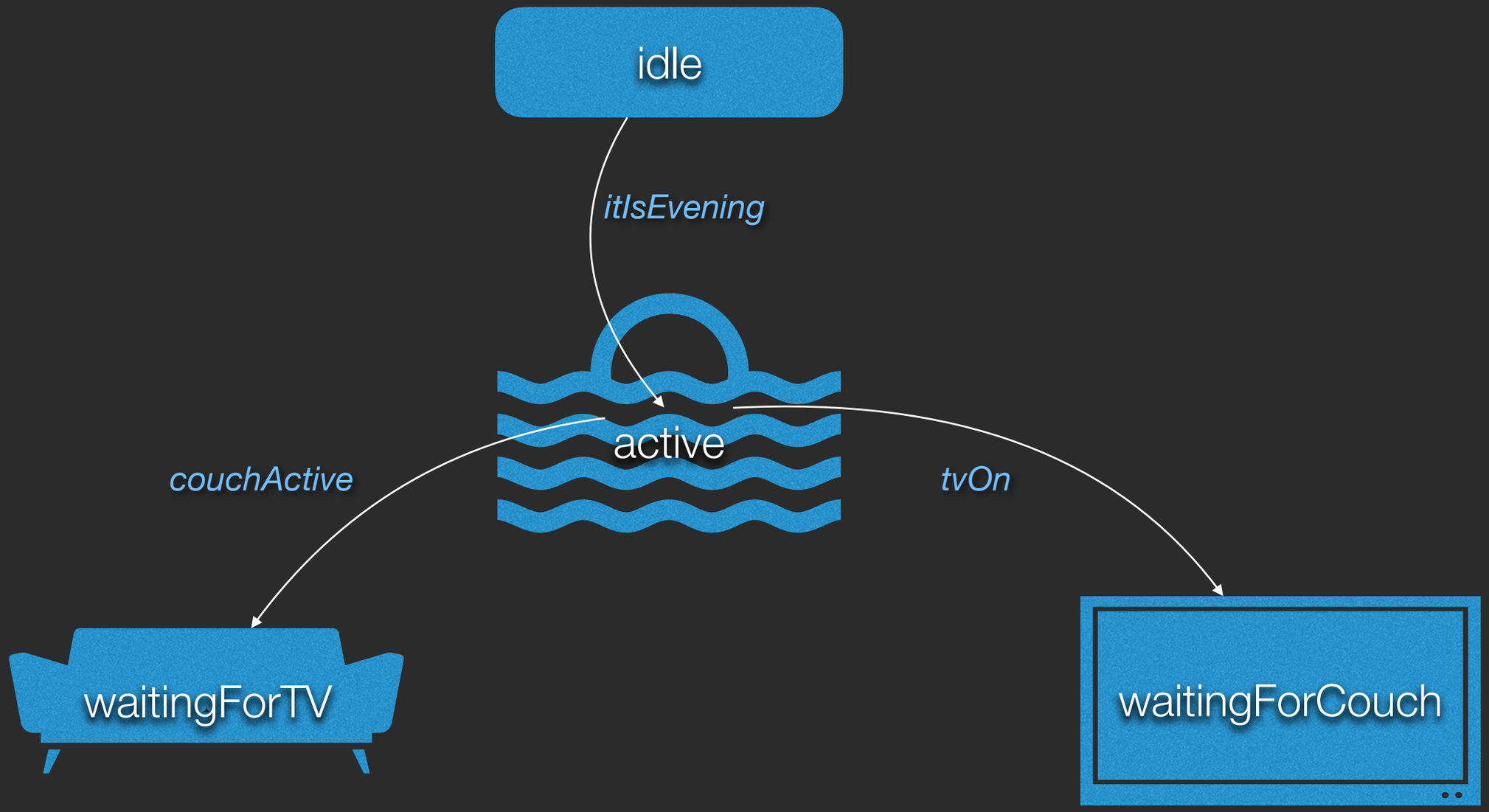
# Home automation

# Home automation

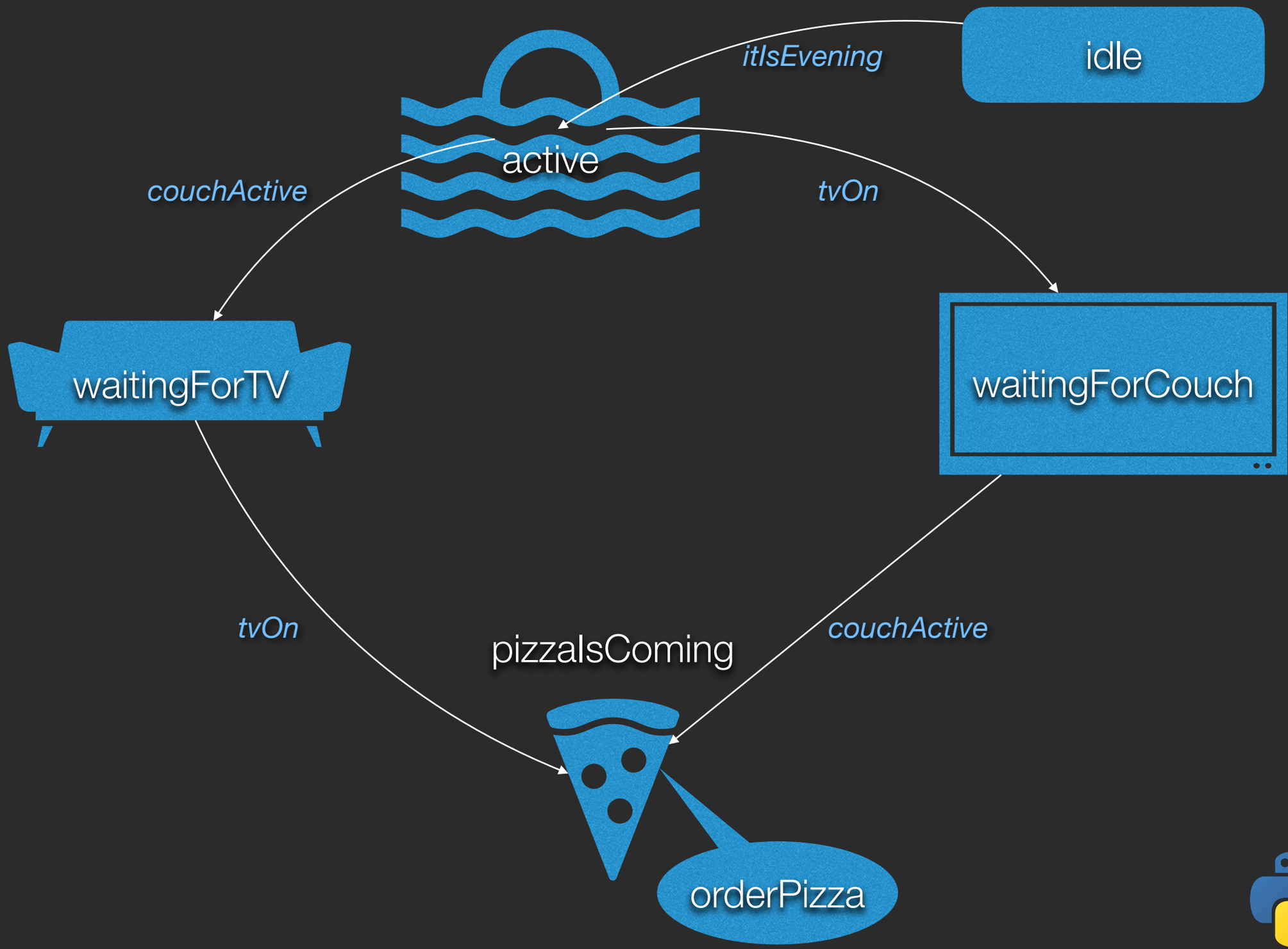
idle

# Home automation

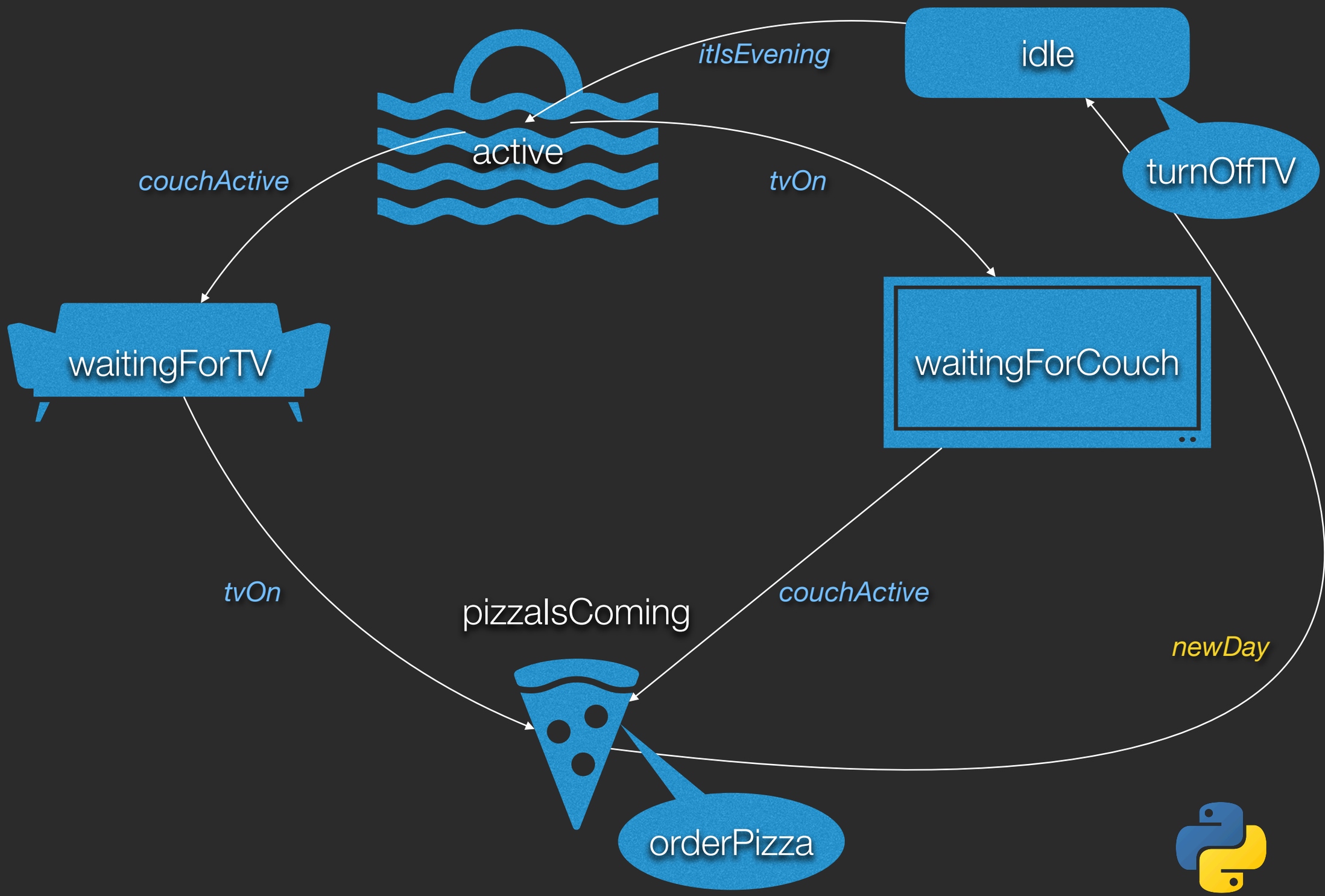

# Home automation




# Home automation



# Home automation



# Semantic Model

```python
@dataclass
class Message:
    name: str
    code: str

class Event(Message):
    pass

class Action(Message):
    pass
```

```python
@dataclass
class Transition:
    source: State
    destination: State
    trigger: Event

@dataclass
class State:
    name: str
    actions: List [Action] = field(default_factory=list)
    transitions: Dict [Event, Transition] = field(default_factory=dict)

@dataclass
class StateMachine:
    initial_state: State
    current_state: State
    reset_events: List [State]
```

# Imperative initialization

```
tv_on = Event('tvOn', 'TVON')
couch_active = Event('couchActive', 'COAC')
it_is_evening = Event('itIsEvening', 'ITEV')
new_day = Event('newDay', 'NEDY')

turn_off_tv = Action('turnOffTV', 'TTOF')
order_pizza = Action('orderPizza', 'OPZZ')

idle = State('idle')
active = State('active')
waiting_for_couch = State('waitingForCouch')
waiting_for_tv = State(‘waitingForTV')
pizza_is_coming = State('pizzaIsComing')

idle.add_transition(Transition(idle, active, it_is_evening))
idle.add_actions(turn_off_tv)

active.add_transition(Transition(active, waiting_for_couch, tv_on))
active.add_transition(Transition(active, waiting_for_tv, couch_active))

waiting_for_couch.add_transition(Transition(waiting_for_couch, pizza_is_coming,
couch_active))

waiting_for_tv.add_transition(Transition(waiting_for_tv, pizza_is_coming, tv_on))

pizza_is_coming.add_transition(Transition(pizza_is_coming, idle, new_day))
pizza_is_coming.add_actions(order_pizza)

machine = StateMachine(idle, idle, {new_day})
```

# YAML

```yaml
events:
  -
    name: tvOn
    code: TVON
  -
    name: itIsEvening
    code: EHAC
  -
    name: couchActive
    code: COAC
  -
    name: newDay
    code: NEDY

resetEvents:
  - newDay

commands:
  -
    name: turnOffTV
    code: TTOF
  -
    name: orderPizza
    code: OPZZ
```

```yaml
states:
  -
    name: idle
    actions:
      - turnOffTV
    transitions:
      -
        trigger: itIsEvening
        target: active
  - name: active
      transitions:
      -
        trigger: couchActive
        target: waitingForTV
      -
        trigger: tvOn
        target: waitingForCouch
-
  name: waitingForCouch
  transitions:
    -
      trigger: couchActive
      target: pizzaIsComing
-
  name: waitingForTV
  transitions:
    -
      trigger: tvOn
      target: pizzaIsComing
-
  name: pizzaIsComing
  actions:
    - orderPizza
```

# External DSL

```
events
    tvOn          TVON
    itIsEvening   EHAC
    couchActive   COAC
    newDay        NEDY
end

resetEvents
    newDay
end

commands
    turnOffTV     TTOF
    orderPizza    OPZZ
end

state idle
    actions {turnOffTV}
    itIsEvening => active
end
```

```
state active
    tvOn => waitingForCouch
    couchActive => waitingForTV
end

state waitingForCouch
  couchActive => pizzaIsComing
end

state waitingForTV
    tvOn => pizzaIsComing
end

state pizzaIsComing
    actions {orderPizza}
    newDay => idle
end
```

# Internal DSL

```
with events:
    tv_on = 'TVON'
    it_is_evening = 'EHAC'
    couch_active = 'COAC'
    new_day = 'NEDY'

reset_events(new_day)

with commands:
    turn_off_tv = 'TTOF'
    order_pizza = 'OPZZ'

with initial_state("idle"):
    actions(turn_off_tv)
    transitions[it_is_evening:'active']

with state('active'):
    transitions[
        tv_on:'waitingForCouch',
        couch_active:'waitingForTV']

with state('waitingForTV'):
    transitions[couch_active:'pizzaIsComing']

with state('waitingForCouch'):
    transitions[tv_on:'pizzaIsComing']

with state('pizzaIsComing'):
    actions(order_pizza)
```

# Domain-specific language

Is a computer programming language of limited expressiveness focused on a particular domain.

# Domain-specific language

Is a computer programming language of limited expressiveness focused on a particular domain.

# Domain-specific language

Is a computer programming language of limited expressiveness focused on a particular domain.

# Why DSL

- Increase productivity
- Improve communication with non-technical people
- Make reasoning about the system easier
- Work around limitations of target system

# DSL Zoo

## RegEx

```
^((?=\S*?[A-Z])(?=\S*?[a-z])(?=\S*?[0-9]).{6,}\S$
```

Password validation rule:
Checks that a password has a minimum of 6 characters, at least 1 uppercase letter, 1 lowercase letter, and 1 number with no spaces.

## Rake

```
namespace :cake do
  desc 'make pancakes'
  task :pancake => [:flour,:milk,:egg,:baking_powder] do
    puts "sizzle"
  end
  task :butter do
    puts "cut 3 tablespoons of butter into tiny squares"
  end
  task :flour => :butter do
    puts "use hands to knead butter squares into 1{{frac|1|2}} cup flour"
  end
  task :milk do
    puts "add 1{{frac|1|4}} cup milk"
  end
  task :egg do
    puts "add 1 egg"
  end
  task :baking_powder do
    puts "add 3{{frac|1|2}} teaspoons baking powder"
  end
end
```

```
<html>
<head>
    <style>
        body {
            background-image: url("https://upload.wikimedia.org/wikipedia/
commons/thumb/c/c3/Python-logo-notext.svg/1000px-Python-logo-
notext.svg.png");
            background-color: #2B2B2B;
        }

        h1 {
            color: #FFD759;
            text-align: center;
        }

        p {
            font-family: verdana;
            font-size: 20px;
            color: #FFD759
        }
    </style>
</head>
<body>
<h1>Hello Pycon!</h1>
</body>
</html>
```


## attrs

```
@attr.s
class States:
    name: attr.ib()
    actions: attr.ib(default=attr.Factory(list))
    transitions: attr.ib(default=attr.Factory(dict))
```

## ANTLR

```
parser grammar ANTLRv4Parser;

options
   { tokenVocab = ANTLRv4Lexer; }

// The main entry point for parsing a v4 grammar.
grammarSpec
   : DOC_COMMENT* grammarType identifier SEMI prequelConstruct* rules modeSpec* EOF
   ;

grammarType
   : (LEXER GRAMMAR | PARSER GRAMMAR | GRAMMAR)
   ;

// This is the list of all constructs that can be declared before
// the set of rules that compose the grammar, and is invoked 0..n
// times by the grammarPrequel rule.
prequelConstruct
   : optionsSpec
   | delegateGrammars
   | tokensSpec
   | channelsSpec
   | action
   ;

// ------------
// Options - things that affect analysis and/or code generation
optionsSpec
   : OPTIONS LBRACE (option SEMI)* RBRACE
   ;

option
   : identifier ASSIGN optionValue
   ;
```

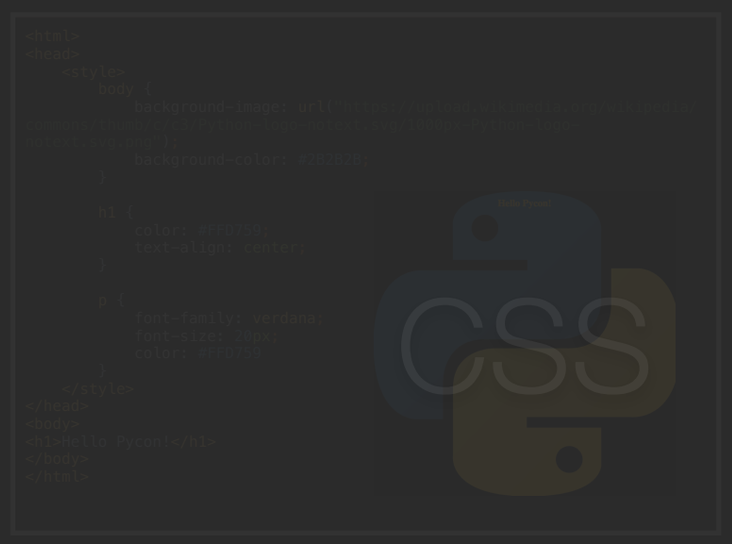
DSL Zoo

RegEx

Rake

```
^((?=\S*?[A-Z])(?=\S*?[a-z])(?=\S*?[0-9]).{6,})\S$
```

Password validation rule:
Checks that a password has a minimum of 6 characters, at least 1 uppercase letter, 1 lowercase letter, and 1 number with no spaces.




attrs

```
@attr.s
class States:
    name: attr.ib()
    actions: attr.ib(default=attr.Factory(list))
    transitions: attr.ib(default=attr.Factory(dict))
```

ANTLR

```
parser grammar ANTLRv4Parser;

options
   { tokenVocab = ANTLRv4Lexer; }

// The main entry point for parsing a v4 grammar.
grammarSpec
   : DOC_COMMENT* grammarType identifier SEMI prequelConstruct* rules modeSpec* EOF
   ;

grammarType
   : (LEXER GRAMMAR | PARSER GRAMMAR | GRAMMAR)
   ;

// This is the list of all constructs that can be declared before
// the set of rules that compose the grammar, and is invoked 0..n
// times by the grammarPrequel rule.
prequelConstruct
   : optionsSpec
   | delegateGrammars
   | tokensSpec
   | channelsSpec
   | action
   ;

// ------------
// Options - things that affect analysis and/or code generation
optionsSpec
   : OPTIONS LBRACE (option SEMI)* RBRACE
   ;

option
   : identifier ASSIGN optionValue
   ;
```

# DSL Zoo Rake

```
namespace :cake do
  desc 'make pancakes'
  task :pancake => [:flour,:milk,:egg,:baking_powder] do
    puts "sizzle"
  end
  task :butter do
    puts "cut 3 tablespoons of butter into tiny squares"
  end
  task :flour => :butter do
    puts "use hands to knead butter squares into 1{{frac|1|2}} cup flour"
  end
  task :milk do
    puts "add 1{{frac|1|4}} cup milk"
  end
  task :egg do
    puts "add 1 egg"
  end
  task :baking_powder do
    puts "add 3{{frac|1|2}} teaspoons baking powder"
  end
end
```

# DSL Zoo CSS

```
<html>
<head>
    <style>
        body {
            background-image: url("https://upload.wikimedia.org/wikipedia/commons/
thumb/c/c3/Python-logo-notext.svg/1000px-Python-logo-notext.svg.png");
            background-color: #2B2B2B;
        }

        h1 {
            color: #FFD759;
            text-align: center;
        }

        p {
            font-family: verdana;
            font-size: 20px;
            color: #FFD759
        }
    </style>
</head>
<body>
<h1>Hello Pycon!</h1>
</body>
</html>
```

# DSL Zoo ANTLR

```
parser grammar ANTLRv4Parser;

options
   { tokenVocab = ANTLRv4Lexer; }

// The main entry point for parsing a v4 grammar.
grammarSpec
   : DOC_COMMENT* grammarType identifier SEMI prequelConstruct* rules modeSpec* EOF
   ;

grammarType
   : (LEXER GRAMMAR | PARSER GRAMMAR | GRAMMAR)
   ;

// This is the list of all constructs that can be declared before
// the set of rules that compose the grammar, and is invoked 0..n
// times by the grammarPrequel rule.
prequelConstruct
   : optionsSpec
   | delegateGrammars
   | tokensSpec
   | channelsSpec
   | action
   ;

// ------------
// Options - things that affect analysis and/or code generation
optionsSpec
   : OPTIONS LBRACE (option SEMI)* RBRACE
   ;

option
   : identifier ASSIGN optionValue
   ;
```

```
^((?=\S*?[A-Z
```

Password vali
Checks that a
characters, at
letter, and 1 n

```
<html>
<head>
    <style>
        body {
            background-image: url(
commons/thumb/c/c3/Python-logo-not
notext.svg.png");
            background-color: #2B2
        }

        h1 {
            color: #FFD759;
            text-align: center;
        }

        p {
            font-family: verdana;
            font-size: 20px;
            color: #FFD759
        }
    </style>
</head>
<body>
<h1>Hello Pycon!</h1>
</body>
</html>
```

```
er] do


ares"


{frac|1|2}} cup flour"
```

DSL Zoo

attrs

RegEx

Rake

```
@attr.s
class States:
    name: attr.ib()
    actions: attr.ib(default=attr.Factory(list))
    transitions: attr.ib(default=attr.Factory(dict))
```

CSS

ANTLR

# Why Internal DSL

- Easier to start with
- Host language does a lot of work for you
- Tools support from the host language.
- Ability to invoke/create complex logic

# Internal DSL objectives

- Provide an interface that feels like a language expressed in terms of your domain

- Reduce syntactic noise

# Python Internal DSL techniques

# Fluid interface layer

A thin layer of functions and Builder-type objects, that helps us to ‘wire up’ the underlying model

# Fluid interface layer

```python
@dataclass
class Message:
    name: str
    code: str

class Event(Message):
    pass

class Action(Message):
    pass
```

```python
@dataclass
class Transition:
    source: State
    destination: State
    trigger: Event

@dataclass
class State:
    name: str
    actions: List [Action] = field(default_factory=list)
    transitions: Dict [Event, Transition] = field(default_factory=dict)

@dataclass
class StateMachine:
    initial_state: State
    current_state: State
    reset_events: List [State]
```

# Fluid interface layer

```python
@dataclass
class StateBuilder:
    state: State

    def on(self, trigger):
        return TransitionBuilder(self, trigger)

    def actions(self, *args):
        self.state.add_actions(*args)
        return self
```

```python
@dataclass
class TransitionBuilder:
    state_builder: StateBuilder
    trigger: Event

    def transition_to(self, target: StateBuilder):
        constructed_state = self.state_builder.state
        constructed_state.add_transition(Transition(constructed_state, target.state, self.trigger))
        return self.state_builder


def state(name=None, actions=None):
    return StateBuilder(State(name=name or uuid4(), actions=actions or []))


def state_machine(*, initial_state: StateBuilder, reset_events):
    return StateMachine(initial_state.state, initial_state.state, reset_events)
```

# Fluid interface layer

```
tv_on = Event('TVON')
couch_active = Event('couchActive', 'COAC')
it_is_evening = Event('itIsEvening', 'ITEV')
new_day = Event('newDay', 'NEDY')

turn_off_tv = Action('turnOffTV', 'TTOF')
order_pizza = Action('orderPizza', 'OPZZ')

idle = State(‘idle')
active = State('active')
waiting_for_couch = State('waitingForCouch')
waiting_for_tv = State(‘waitingForTV')
pizza_is_coming = State('pizzaIsComing')

idle.add_transition(Transition(idle, active, it_is_evening))
idle.add_actions(turn_off_tv)

active.add_transition(Transition(active, waiting_for_couch, tv_on))
active.add_transition(Transition(active, waiting_for_tv, couch_active))

waiting_for_couch.add_transition(Transition(waiting_for_couch, pizza_is_coming,
couch_active))

waiting_for_tv.add_transition(Transition(waiting_for_tv, pizza_is_coming, tv_on))

pizza_is_coming.add_transition(Transition(pizza_is_coming, idle, new_day))
pizza_is_coming.add_actions(order_pizza)

machine = StateMachine(idle, idle, {new_day})
```

# Fluid interface layer

```
tv_on = Event(code='TVON')
couch_active = Event(code='COAC')
it_is_evening = Event(code='ITEV')
new_day = Event(code='NEDY')

turn_off_tv = Action(code='TTOF')
order_pizza = Action(code='OPZZ')

pizza_is_coming = (state(
    actions=[order_pizza]
))

waiting_for_tv = (state()
                  .on(tv_on).transition_to(pizza_is_coming))

waiting_for_couch = (state()
                     .on(couch_active).transition_to(pizza_is_coming))

active = (state()
          .on(couch_active).transition_to(waiting_for_tv)
          .on(tv_on).transition_to(waiting_for_couch))

idle = (state(actions=[turn_off_tv])
        .on(it_is_evening).transition_to(active))

machine = state_machine(initial_state=idle, reset_events={new_day})
```

# Creative usage of the language syntax

# Creative usage of the language syntax


# Context Managers (with keyword)

Allows you to execute code before and after provided block of code

# Context Managers (with keyword)

Allows you to execute code before and after provided block of code

```
@contextmanager
def tag(tag_name):
    print(f'<{tag_name}>')
    try:
        yield
    finally:
        print(f'</{tag_name}>')
```

# Context Managers (with keyword)

Allows you to execute code before and after provided block of code

```
@contextmanager
def tag(tag_name):
    print(f'<{tag_name}>')
    try:
        yield
    finally:
        print(f'</{tag_name}>')


with tag("html"):
    with tag("body"):
        with tag("h1"):
            print("Whee")
```

# Context Managers (with keyword)

```
@contextmanager
def tag(tag_name):
    print(f'<{tag_name}>')
    try:
        yield
    finally:
        print(f'</{tag_name}>')

with tag("html"):
    with tag("body"):
        with tag("h1"):
            print("Whee")

# prints:

<html>
<body>
<h1>
Whee
</h1>
</body>
</html>
```

# Context Managers (with keyword)

Allows you to execute code before and after provided block of code

```
with path('/land/of/magic/binaries',
          behavior=PREPEND_BEHAVIOUR):
    with cd('/tmp'):
        run('touch PyCon')
```

# Context Managers (with keyword)

Allows you to execute code before and after provided block of code

```
@contextmanager
def cd(new_dir):
    current_dir = getcwd()
    chdir(new_dir)

    try:
        yield
    finally:
        chdir(current_dir)


with path('/land/of/magic/binaries',
          behavior=PREPEND_BEHAVIOUR):
    with cd('/tmp'):
        run('touch PyCon')
```

# Context Managers (with keyword)

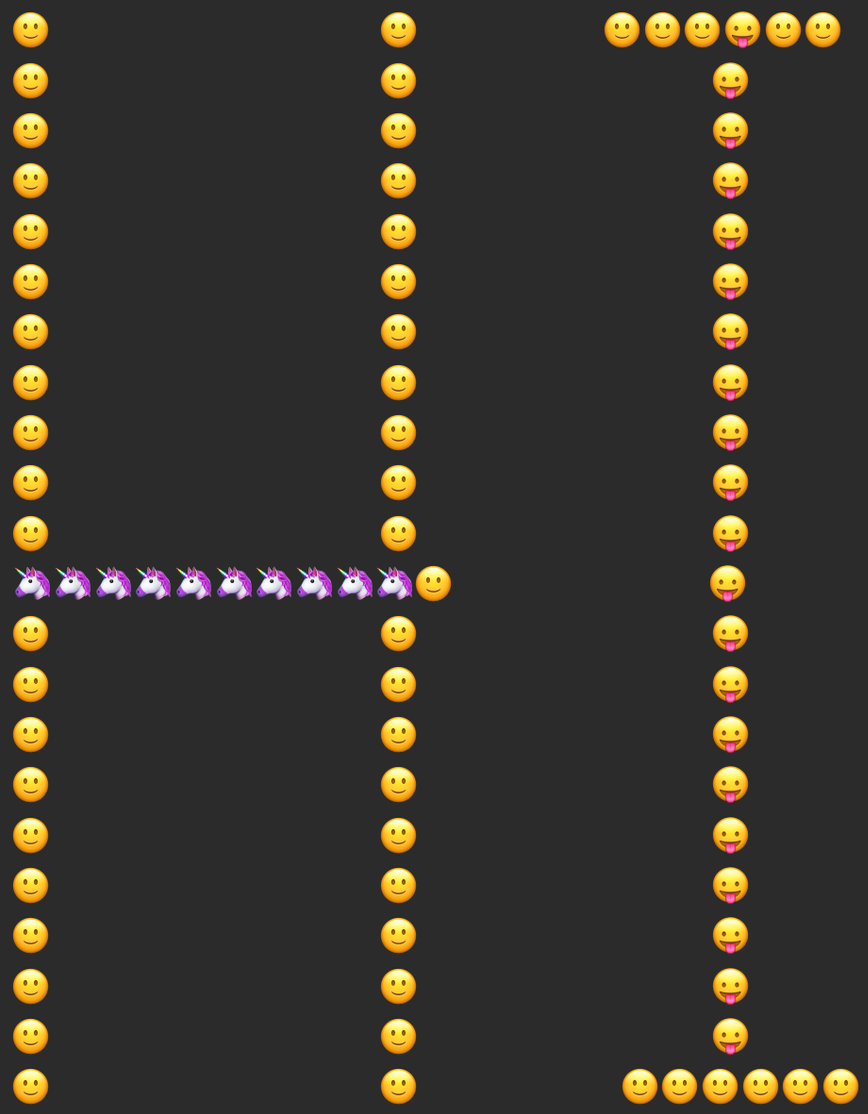
Allows you to execute code before and after provided block of code

```
@contextmanager
def path(path_to_add, behavior=PREPEND_BEHAVIOUR):
    path_copy = environ.get(PATH_VAR, "")
    if behavior == PREPEND_BEHAVIOUR:
        environ[PATH_VAR] = f"{path_to_add}:{path_copy}"
    try:
        yield
    finally:
        environ[PATH_VAR] = path_copy


    with path('/land/of/magic/binaries',
              behavior=PREPEND_BEHAVIOUR):
        with cd('/tmp'):
            run('touch PyCon')
```

# Magic methods overloading

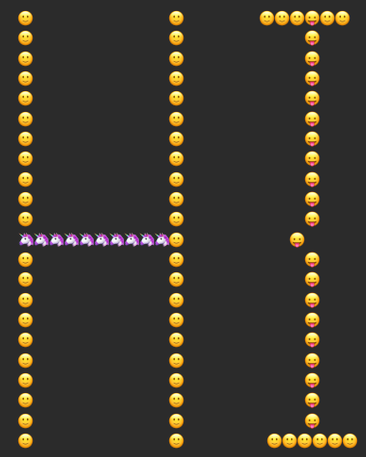
An implicit object behavior is driven by magic methods.
That can be redefined to suit your language needs

# Magic methods overloading
# Character Canvas DSL



# Magic methods overloading

## Character Canvas DSL

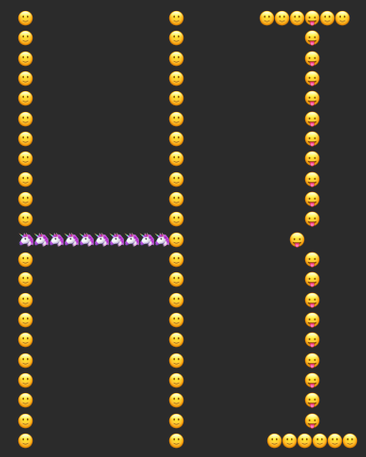


```
p = CharacterCanvas('🙂', '  ')

def h():
    p // letter_height
    p ^ (letter_height / 2)

    with p('🦄'):
        p > letter_width

    p // (letter_height / 2)
    p ^ letter_height


def space():
    p > 1
    with p('  '):
        p > letter_width / 2
```

```
def i():
    p > letter_width / 2
    p < letter_width / 4

    with p('😛'):
        p // letter_height

    p < letter_width / 4
    p > letter_width / 2 + 1
```

# Magic methods overloading

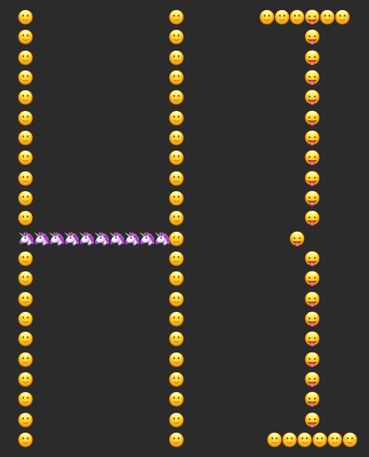
## Character Canvas DSL



```python
@dataclass
class CharacterCanvas:
    symbol: str = '*'
    filler: str = ' '
    position: Point = Point()
    #...

    def transform(self, amount, transformation: Callable[[Point], Point]):
        for _ in range(int(amount)):
            self.content[self.position] = self.symbol
            self.position = transformation(self.position)

        self.content[self.position] = self.symbol

    def __lt__(self, other):
        self.transform(other, lambda point: Point(point.x - 1, point.y))

    def __gt__(self, other):
        self.transform(other, lambda point: Point(point.x + 1, point.y))

    #...
```

# Magic methods overloading

## Character Canvas DSL



```python
@dataclass
class CharacterCanvas:
    symbol: str = '*'
    filler: str = ' '
    position: Point = Point()
    #...

    def transform(self, amount, transformati
        for _ in range(int(amount)):
            self.content[self.position] = se
            self.position = transformation(s

        self.content[self.position] = self.s

    def __lt__(self, other):
        self.transform(other, lambda point:

    def __gt__(self, other):
        self.transform(other, lambda point:

    #...
```

```python
#...
def __call__(self, contextual_symbol):
    self.contextual_symbol = contextual_symbol
    return self

def __enter__(self):
    self.symbol_stack.append(self.symbol)
    self.symbol = self.contextual_symbol

def __exit__(self, *args):
    self.symbol = self.symbol_stack.pop()
```

# Python is dynamic

# Python is dynamic. Like really



# Set global execution context

```
def achieve_zen():
    import this


the_answer = 42

namespace = dict(achieve_zen=achieve_zen,
                 the_answer=42)

source = Path('global_context_manipulation_dsl.py').read_text()
exec(source, namespace)

print(namespace['random_string'])
```


# Set global execution context

```
def achieve_zen():
    import this


the_answer = 42

namespace = dict(achieve_zen=achieve_zen,
                 the_answer=42)

source = Path('global_context_manipulation_dsl.py').read_text()
exec(source, namespace)

print(namespace['random_string'])
```

```
achieve_zen()

print(the_answer)

random_string = "Wheeeee!"
```


# Set global execution context

```
def achieve_zen():
    import this


the_answer = 42

namespace = dict(achieve_zen=achieve_zen,
                 the_answer=42)

source = Path('global_context_manipulation_dsl.py').read_text()
exec(source, namespace)

print(namespace['random_string'])
```

```
achieve_zen()

print(the_answer)

random_string = "Wheeeee!"
```



# Set local execution context

## Emulate implicit self

```
test_dict = {}
with in_context(test_dict):
    this['is'] = 'very'
    update({'useful': 'example', 'idict': {}})
        with in_context(this['idict']):
            update({'internal': 'stuff'})


print(test_dict)
```


# Set local execution context

## Emulate implicit self

```
test_dict = {}
with in_context(test_dict):
    this['is'] = 'very'
    update({'useful': 'example', 'idict': {}})

    with in_context(this['idict']):
        update({'internal': 'stuff'})

    print(test_dict)

# prints:
{'is': 'very', 'useful': 'example', 'idict': {'internal': 'stuff'}}
```


# Set local execution context

## Emulate implicit self

```
@contextmanager
def in_context(context_object):
    decorator_frame = inspect.currentframe().f_back
    caller_frame = decorator_frame.f_back
    caller_locals = caller_frame.f_locals
    locals_snapshot = caller_locals.copy()
    caller_locals.update({field: getattr(context_object, field)
                          for field in dir(context_object)
                          if is_public(field)})

    caller_locals['this'] = context_object
    try:
        yield
    finally:
        caller_locals.clear()
        caller_locals.update(locals_snapshot)
```


# Dynamically create fields/functions/classes

```
@contextmanager
def tag(tag_name):
    print(f'<{tag_name}>')
    try:
        yield
    finally:
        print(f'</{tag_name}>')


with tag("html"):
    with tag("body"):
        with tag("h1"):
            print("Whee")
```


# Dynamically create fields/functions/classes

```
@contextmanager
def tag(tag_name):
    print(f'<{tag_name}>')
    try:
        yield
    finally:
        print(f'</{tag_name}>')


with tag("html"):
    with tag("body"):
        with tag("h1"):
            print("Whee")
```

```
@dataclass
class XmlBuilder:
    tag_name: str = "html"

    def __enter__(self):
        print(f'<{self.tag_name}>')

    def __exit__(self, *args):
        print(f'</{self.tag_name}>')

    def __getattr__(self, item):
        return XmlBuilder(item)

t = XmlBuilder()

with t:
    with t.body:
        with t.h1:
            print("Whee")
```


# Dynamically create fields/functions/classes

```
with path('/land/of/magic/binaries',
          behavior=PREPEND_BEHAVIOUR):
    with cd('/tmp'):
        run('touch PyCon')
```


# Dynamically create fields/functions/classes

```
with path('/land/of/magic/binaries',
          behavior=PREPEND_BEHAVIOUR):
    with cd('/tmp'):
        run('touch PyCon')
```

```
with Path.prepend('/land/of/magic/binaries'):
    with cd('/tmp'):
        c.touch('PyCon')
```


# Dynamically create fields/functions/classes

```
with path('/land/of/magic/binaries',
          behavior=PREPEND_BEHAVIOUR):
    with cd('/tmp'):
        run('touch PyCon')
```

```
def run(command):
    subprocess.call(command, shell=True)
```


# Dynamically create fields/functions/classes

```
with Path.prepend('/land/of/magic/binaries'):
    with cd('/tmp'):
        c.touch('PyCon')

                    def run(command):
                        subprocess.call(command, shell=True)
```

```
@dataclass
class NamedRunner:
    prefix: str = ''

    def __getattr__(self, item):
        self.prefix = item
        return self

    def __call__(self, command):
        run(f'{self.prefix} {command}')
```


# Dynamically create fields/functions/classes

```
with path('/land/of/magic/binaries',
          behavior=PREPEND_BEHAVIOUR):
    with cd('/tmp'):
        run('touch PyCon')
```

```
@contextmanager
def path(path_to_add, behavior=PREPEND_BEHAVIOUR):
    path_copy = environ.get(PATH_VAR, "")

    if behavior == PREPEND_BEHAVIOUR:
        environ[PATH_VAR] = f"{path_to_add}:{path_copy}"
    try:
        yield
    finally:
        environ[PATH_VAR] = path_copy
```


# Dynamically create fields/functions/classes

```
with Path.prepend('/land/of/magic/binaries'):
    with cd('/tmp'):
        c.touch('PyCon')
```

```
class MetaPath(type):
    def __getattr__(cls, item):
        return cls(behavior=item)
```

```
class Path(metaclass=MetaPath):
    def __init__(self, path_to_add='', behavior=PREPEND_BEHAVIOUR):
        self.path_to_add = path_to_add
        self.behavior = behavior

    def __enter__(self):
        self.path_copy = environ.get(PATH_VAR, "")

        if self.behavior == PREPEND_BEHAVIOUR:
            environ[PATH_VAR] = f"{self.path_to_add}:{self.path_copy}"

    def __exit__(self, exc_type, exc_val, exc_tb):
        environ[PATH_VAR] = self.path_copy

    def __call__(self, path_to_add):
        self.path_to_add = path_to_add
        return self
```

# Abstract Syntax Tree modifications

Allows you modify the semantics of any Python syntactic construct.

To extend or completely change it’s functionality.

# AST modifications. Elixir style pipes

pipes library by @robinhilliard allows you to transform bit shift operators (<<, >>) into functional pipes

# AST modifications. Elixir style pipes

pipes library by @robinhilliard allows you to transform bit shift operators (<<, >>) into functional pipes

```
def group_by_character_set(input_string):
    print(group_to_dict(
        groupby(map(lambda x: x.lower(),
                    input_string.split()
                    ),
                lambda x: frozenset(x))
    ))
```

# AST modifications. Elixir style pipes

```
def group_by_character_set(input_string):
    print(group_to_dict(
        groupby(map(lambda x: x.lower(),
                    input_string.split()
                    ),
                lambda x: frozenset(x))
    ))


@pipes
def expression_with_pipes(input_string):
    input_string.split() << \
    map(lambda x: x.lower()) >> \
    groupby(lambda x: frozenset(x)) >> \
    group_to_dict >> print
```

# AST modifications. Macropy

macropy is a framework to simplify import time AST modifications

# AST modifications. Macropy. PINQ to SQL

## PINQ - Python INtegrated Query.

```
query = sql[(
    x.name for x in db.country
    if x.gnp / x.population > (
        y.gnp / y.population for y in db.country
        if y.name == 'United Kingdom'
    ).as_scalar()
    if (x.continent == 'Europe')
)]
```

# AST modifications. Macropy. PINQ to SQL

## PINQ get's converted to SqlAlchemy query.

```
query = sql[(
    x.name for x in db.country
    if x.gnp / x.population > (
        y.gnp / y.population for y in db.country
        if y.name == 'United Kingdom'
    ).as_scalar()
    if (x.continent == 'Europe')
)]


query = select([db.country.c.name]).where(
    db.country.c.gnp / db.country.c.population > select(
        [(db.country.c.gnp / db.country.c.population)]
    ).where(
        db.country.c.name == 'United Kingdom'
    ).as_scalar()
).where(
    db.country.c.continent == 'Europe'
)
```

# AST modifications. Macropy. PINQ to SQL

```
query = sql[(
    x.name for x in db.country
    if x.gnp / x.population > (
        y.gnp / y.population for y in db.country
        if y.name == 'United Kingdom'
    ).as_scalar()
    if (x.continent == 'Europe')
)]
```

```
query = select([db.country.c.name]).where(
    db.country.c.gnp / db.country.c.population > select(
        [(db.country.c.gnp / db.country.c.population)]
    ).where(
        db.country.c.name == 'United Kingdom'
    ).as_scalar()
).where(
    db.country.c.continent == 'Europe'
)
```

```
SELECT country_1.name
FROM country AS country_1
WHERE country_1.gnp / country_1.population > (SELECT country_2.gnp / country_2.population
AS anon_1
FROM country AS country_2
WHERE country_2.name = ?) AND country_1.continent = ?
```

# Another look at home automation

```
with events:
    tv_on = 'TVON'
    it_is_evening = 'EHAC'
    couch_active = 'COAC'
    new_day = 'NEDY'

reset_events(new_day)

with commands:
    turn_off_tv = 'TTOF'
    order_pizza = 'OPZZ'

with initial_state("idle"):
    actions(turn_off_tv)
    transitions[it_is_evening:'active']

with state('active'):
    transitions[
        tv_on:'waitingForCouch',
        couch_active:'waitingForTV']

with state('waitingForTV'):
    transitions[couch_active:'pizzaIsComing']

with state('waitingForCouch'):
    transitions[tv_on:'pizzaIsComing']

with state('pizzaIsComing'):
    actions(order_pizza)
```

Do I have to do all those scary/weird things to Python to start creating internal DSLs?


NOPE.

# An accidental DSL

```yaml
name: getFile
description: Retrieves a specified file content.
parameters:
  lineLimit:
    type: String
    default: "0"
    description: >
      Maximum number of lines to return.
      Please specify 0 for no limit.
  filePath:
    type: String
    description: Path to file to retrieve.
command_type: python
command_file: get_file.py
```

# An accidental DSL

```
definition = PythonDefinition(
    name='getFile',
    description='Retrieves a specified file
content',
    command_file_name='get_file.py',
    user='root',
    metadata=[category('OS'), access_level(2)],
    parameters=[entities_limit(), file_path()]
)
```

# Live Demo


Go Forth and build delightfully weird (and amazingly useful) things with Python!

# Go Forth and build delightfully weird (and amazingly useful) things with Python!

# References, code and other nice things

goo.gl/7ZCSiY

root@stvad.org