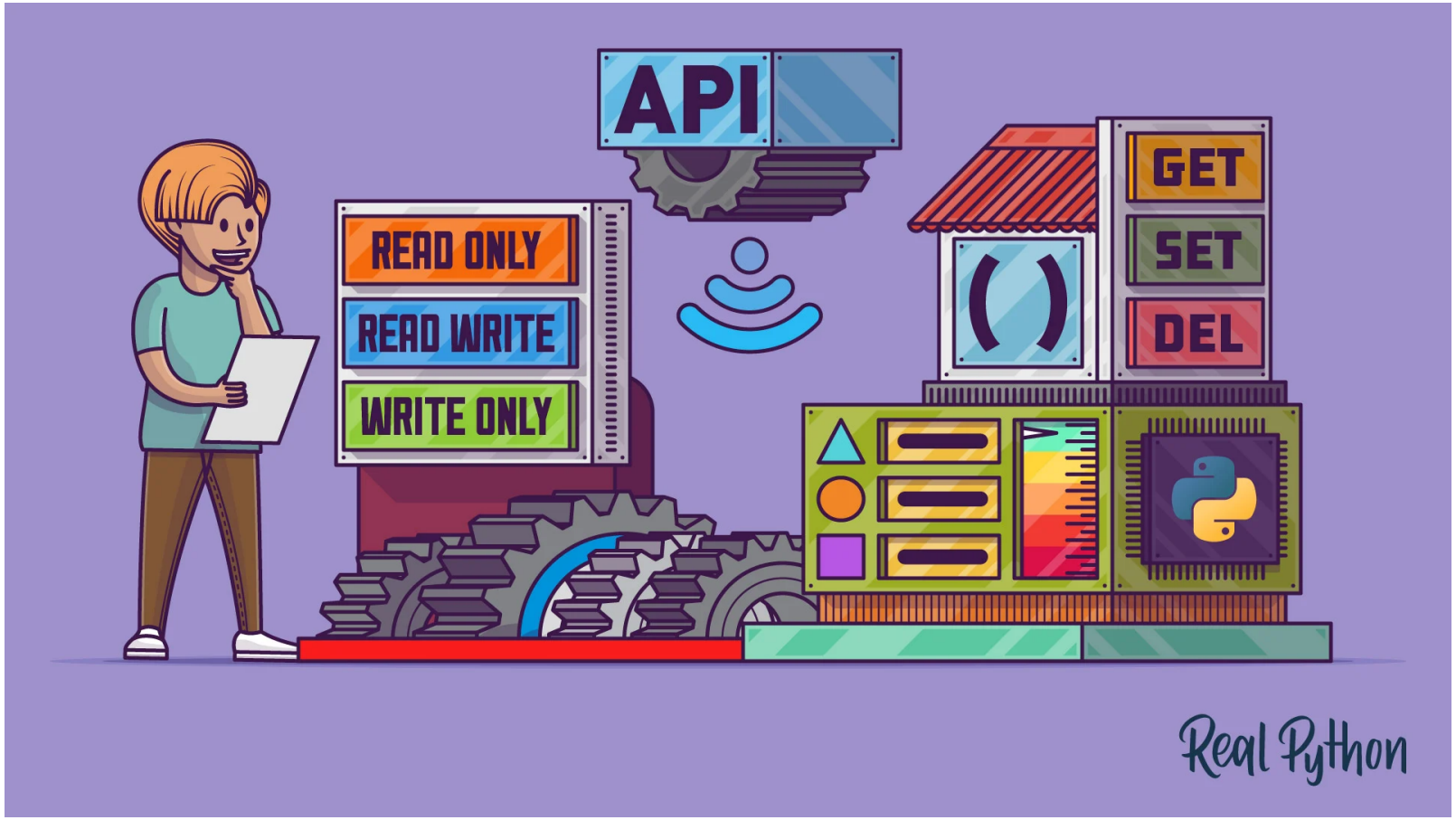
Managing Attributes With Python's Property()



Managing Attributes With Python's Property()

Managing Attributes With Python's Property()

- Create Managed Attributes or Properties in Your Classes
- Perform Lazy Attribute Evaluation and Provide Computed Attributes
- Avoid Setter and Getter Methods to Make Your Classes More Pythonic
- Create Read-Only, Read-Write, and Write-Only Properties
- Create Consistent and Backward-Compatible APIs for Your Classes

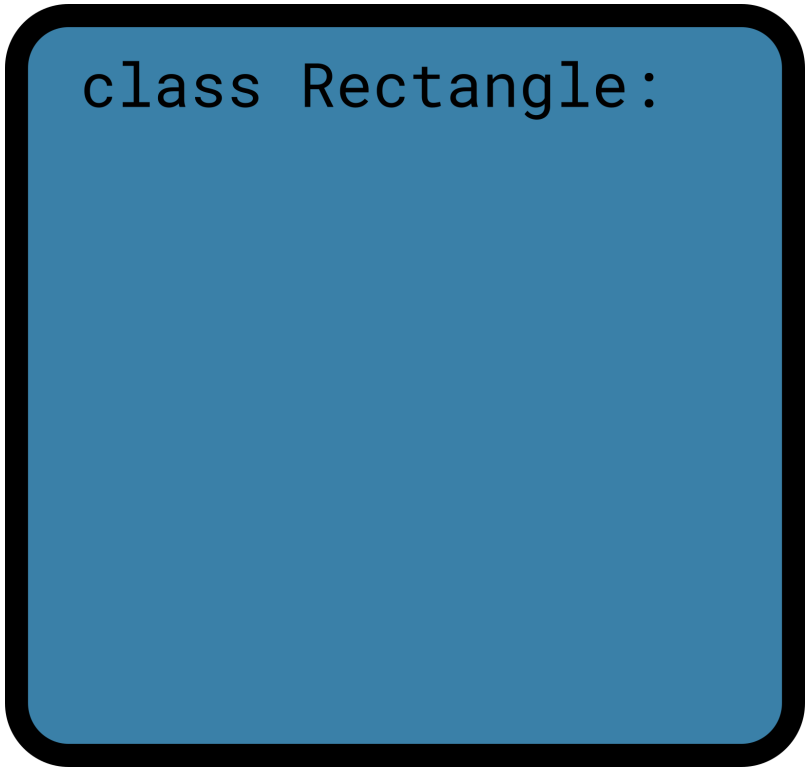
Let’s Get Started!

Managing Attributes With Python's Property()

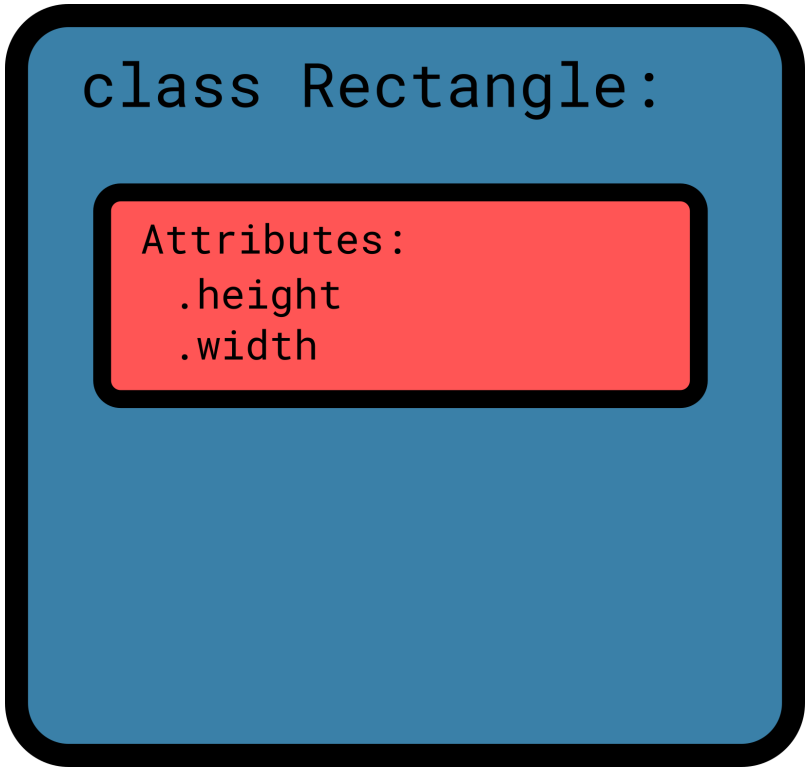
1. **Managing Attributes in Your Classes**
2. Getting Started With `property()`
3. Providing Read-Only Attributes
4. Providing Read-Write Attributes
5. Providing Write-Only Attributes
6. `property()` In Action
7. Overriding Properties in Sub-Classes

Managing Attributes in Your Classes

Managing Attributes in Your Classes



Managing Attributes in Your Classes



Managing Attributes in Your Classes

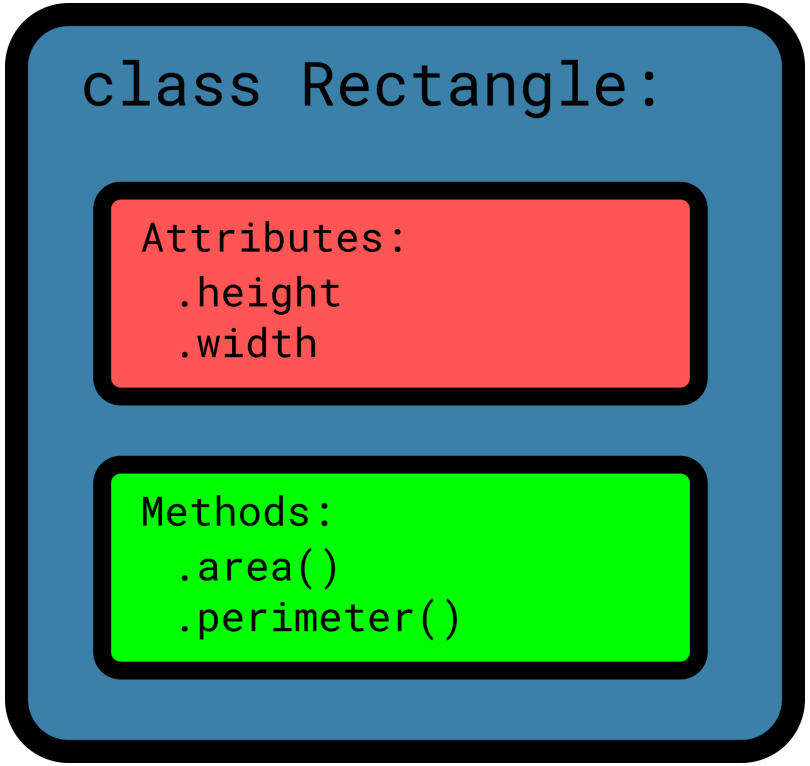
Direct Access:

```
my_rectangle.width = 42
```

Method Use:

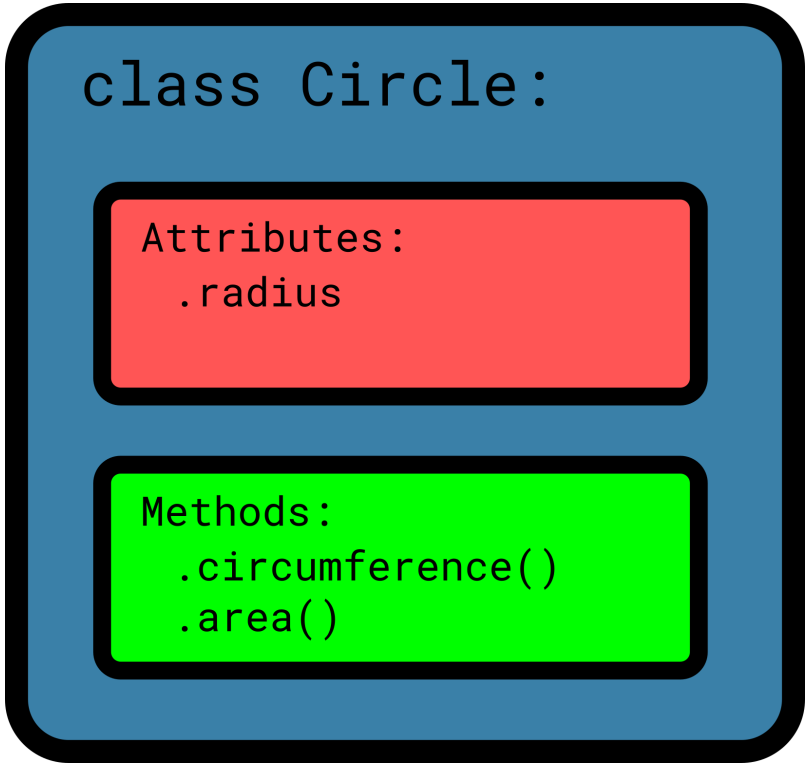
```
my_rectangle.set_width(42)
```

Public API Access

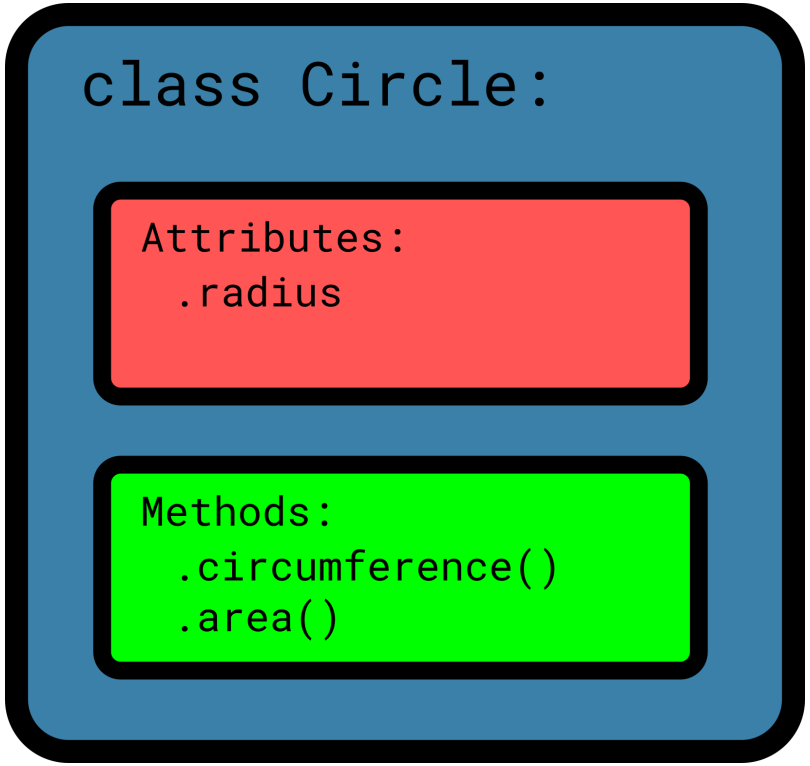



```
>>> from rectangle import Rectangle
>>> my_rect = Rectangle(21, 7)
>>> print(my_rect.area())
147
>>> my_rect.width = 1
>>> print(my_rect.area())
7
>>> my_rect.height = 10
>>> print(my_rect.perimeter())
22
```

The `Circle` Class

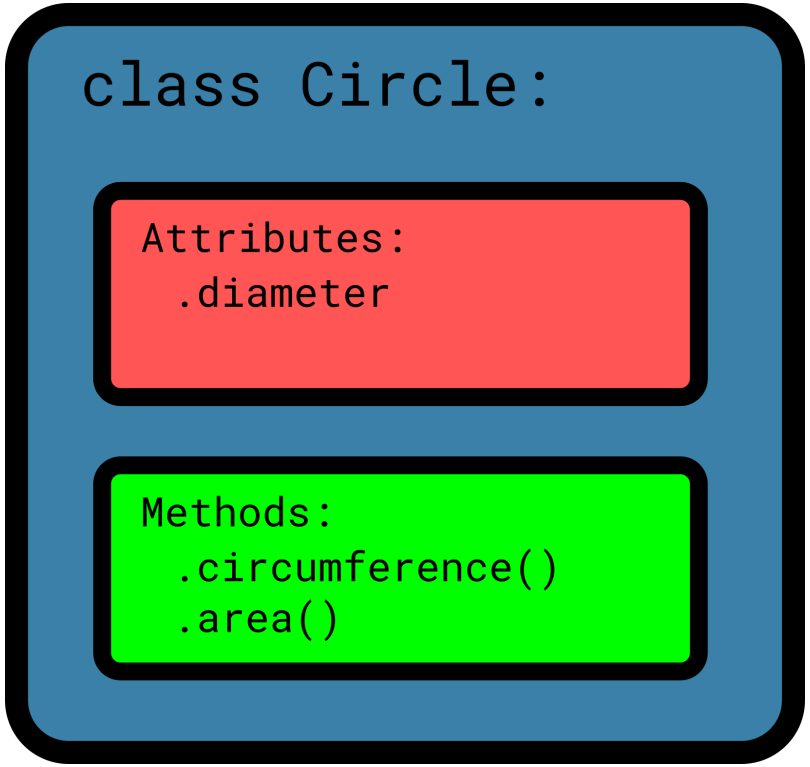


The `Circle` Class

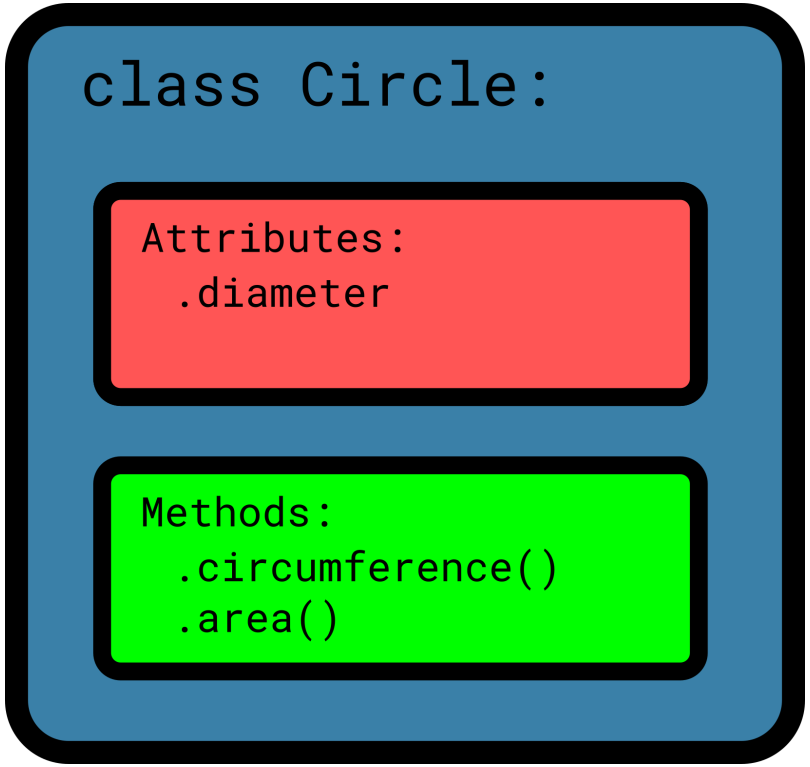



```
>>> from circle import Circle
>>> my_circle = Circle(42)
>>> print(my_circle.radius)
42
>>> print(my_circle.area())
5541.76944
>>> print(my_circle.circumference())
263.893783
```

The `Circle` Class



The `Circle` Class




```
>>> from circle import Circle
>>> my_circle = Circle(42)
>>> print(my_circle.radius)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError:
'Circle' object has no attribute 'radius'
```

Attribute Exposure

- Discouraged In Java and C++
- Getter (Accessor) and Setter (Mutator) Instead
- Implementation Can Be Altered Without Altering API
- Considered By Some To Be Poor Object Oriented Design

The Getter and Setter Approach in Python

Access to Class Members

- Private and Protected Access Modifiers Not Supported
- Python Uses Public and Non-Public Class Members
- A Leading `_` Indicates a Non-Public Attribute or Method
- Just A Convention
    - Access via `class._attribute` Still Possible

The Pythonic Approach

- Previous Code Doesn't Look Pythonic
- Getter and Setter Methods Don't Perform any Processing
- `Point` Can Be Re-Written In a Pythonic Style

Handling Requirement Changes

- Requirements and API Appear Linked
- Implementation Changes Without API Alteration
- Use Python's Properties

Properties

- Combine Elements of Attributes and Methods
- Create Methods That Behave Like Attributes
- Target Attribute Can Be Altered At Any Point

`x` And `y` as Properties

- Continue Access as Attributes
- Underlying Method Allows Internal Modification
- Action Taken Before User Access or Mutation

Properties

- Not Exclusive to Python
  - JavaScript
  - C#
  - Kotlin
- Allow Exposure of Attributes in Public API
- Implementation Changes Are Straightforward

Next: Getting Started With `property()`

Managing Attributes With Python's Property()

Getting Started With `property()`

- The Pythonic Way to Avoid Getter and Setter Methods
- Convert Class Attributes into Properties (Managed Attributes)
- Built-in Function
- Implemented in C

`property()`: Class or Function?

- Commonly Referred to as a Function
- A Class Designed to Work as a Function
- Naming Convention for Classes Not Followed
  - `Property()` vs `property()`

`property()` In Action

- Attach Getter and Setter Methods to Class Attributes
- Handle Internal Implementation Without Exposing Methods
- Specify Deletion Method
- Add a Docstring

`property()` Signature

```
property(fget=None, fset=None, fdel=None, doc=None)
```

- `fget` - Function to Return the Value of the Managed Attribute
- `fset` - Function to Set the Value of the Managed Attribute
- `fdel` - Function to Handle Managed Attribute Deletion
- `doc` - String Representing the Property's Docstring

Attribute Access

- `obj.attr` - `fget()` is Called
- `obj.attr = value` - `fset(value)` is Called
- `del obj.attr` - `fdel()` is Called

Function Objects

- `function` - Function Object
- `function()` - Function Call

Docstrings

- `__doc__` Argument of `property()`
- Readable Using `help(obj.attr)`
- Used in IDEs That Support Docstring Access

Next: Creating Attributes With `property()`

Managing Attributes With Python's Property()

1. Managing Attributes in Your Classes
2. Getting Started With `property()`
   - 2.1 **Creating Attributes With `property()`**
   - 2.2 Using `property()` as a Decorator
3. Providing Read-Only Attributes
4. Providing Read-Write Attributes
5. Providing Write-Only Attributes
6. `property()` In Action
7. Overriding Properties in Sub-Classes

Creating Attributes With `property()`

```
my_object = property(
    fget=_get_attribute,
    fset=_set_attribute,
    fdel=_del_attribute,
    doc="My Lovely Attribute")
```

Bpython Interpreter

https://bpython-interpreter.org/

Properties

- Class Attributes That Manage Instance Attributes
- A Collection of Bundled Methods
- Inspection Reveals Raw Methods
  - `obj.attr.fget`
  - `obj.attr.fset`
  - `obj.attr.fdel`

Python Properties Documentation

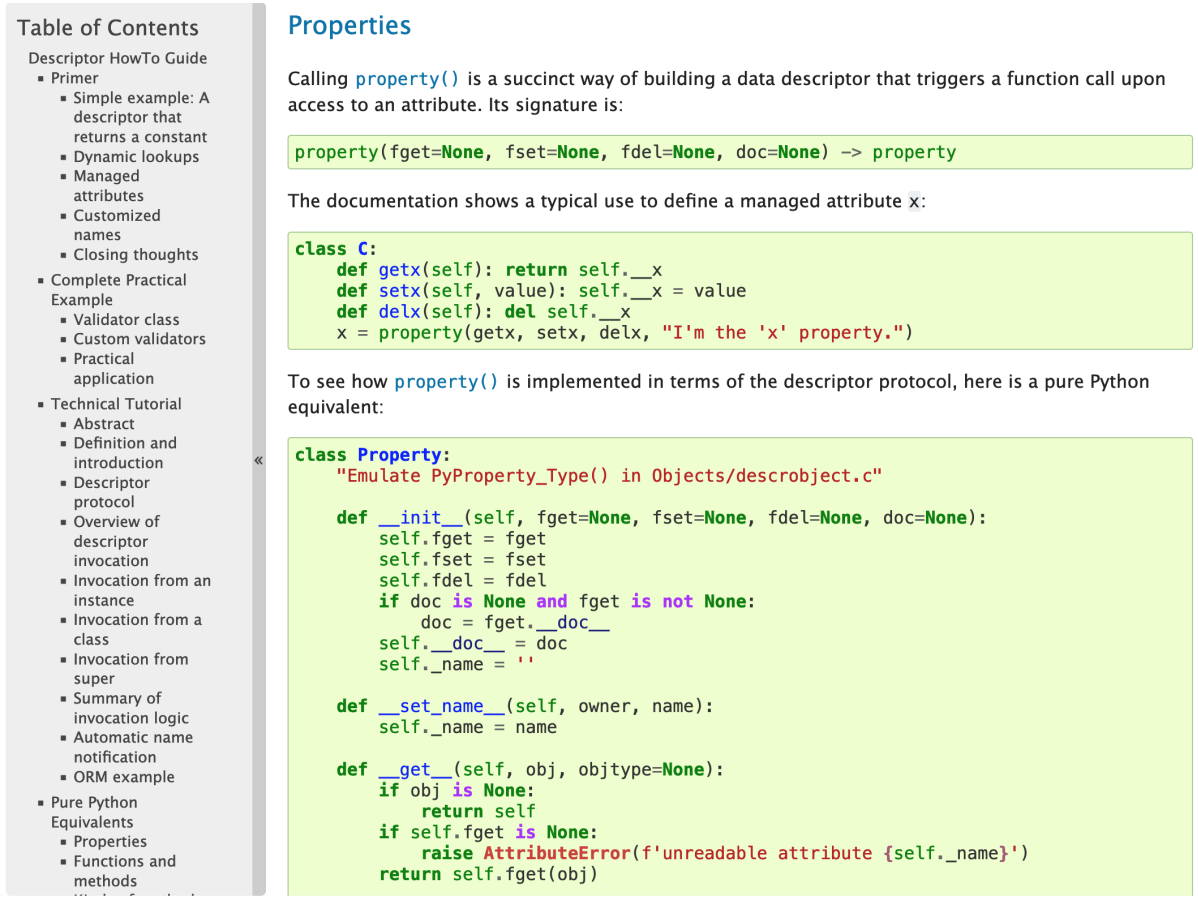


Table of Contents

Properties

Calling `property()` is a succinct way of building a data descriptor that triggers a function call upon access to an attribute. Its signature is:

```
property(fget=None, fset=None, fdel=None, doc=None) -> property
```

The documentation shows a typical use to define a managed attribute `x`:

```
class C:
    def getx(self): return self.__x
    def setx(self, value): self.__x = value
    def delx(self): del self.__x
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

To see how `property()` is implemented in terms of the descriptor protocol, here is a pure Python equivalent:

```
class Property:
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc
        self._name = ''

    def __set_name__(self, owner, name):
        self._name = name

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError(f'unreadable attribute {self._name}')
        return self.fget(obj)
```

https://docs.python.org/3/howto/descriptor.html#properties

Real Python

Next: Using `property()` as a Decorator

Managing Attributes With Python's Property()

Decorators in Python

- Commonly Used
- Functions That Take Another Function
- Add New Functionality
- Attach Pre- and Post-Processing Operations

Decorator Syntax

```
@decorator
def func(a):
    return a
```

Decorator Syntax

```
@decorator
def func(a):
    return a
```

Equivalent To

```
def func(a):
    return a

func = decorator(func)
```

Output from `dir`

```
>>> dir(Circle.radius)
[..., 'deleter', ..., 'getter', 'setter']
```

`@radius.setter`

- Creates a New Property
- Contains Original Property Methods Plus New Setter
- Reassigns Class-Level `.radius` to Hold It

`@property` Recap

- `@property` Decorates Getter Method
- Docstring Must be in Getter Method
- Setter And Deleter Decorated with Getter Name:
  - `@getter.setter`
  - `@getter.deleter`

When Not To Use Properties

- When Getter and Setter Perform No Processing

Drawbacks of Properties When Not Needed

- Verbose
- Confusing
- Slow
- Waste Of Programming Time

Properties Good Practice

- Avoid Explicit Getter and Setter Methods
- Use `@property` Syntax

Next: Providing Read-Only Attributes

Managing Attributes With Python's Property()

1. Managing Attributes in Your Classes
2. Getting Started With `property()`
3. **Providing Read-Only Attributes**
4. Providing Read-Write Attributes
5. Providing Write-Only Attributes
6. `property()` In Action
7. Overriding Properties in Sub-Classes

Providing Read-Only Attributes

Read-Only `Point` Class

Next: Providing Read-Write Attributes

Managing Attributes With Python's Property()

1. Managing Attributes in Your Classes
2. Getting Started With `property()`
3. Providing Read-Only Attributes
4. **Providing Read-Write Attributes**
5. Providing Write-Only Attributes
6. `property()` In Action
7. Overriding Properties in Sub-Classes

Providing Read-Write Attributes

- Getter Method - Read
- Setter Method - Write

Read-Write `Circle` Class

Next: Providing Write-Only Attributes

Managing Attributes With Python's Property()

1. Managing Attributes in Your Classes
2. Getting Started With `property()`
3. Providing Read-Only Attributes
4. Providing Read-Write Attributes
5. **Providing Write-Only Attributes**
6. `property()` In Action
7. Overriding Properties in Sub-Classes

Providing Write-Only Attributes

- Getter Method Raises an Exception

Next: `property()` In Action

Managing Attributes With Python's Property()

6. `property()` **In Action**
   - 6.1 Validating Input Values
   - 6.2 Providing Computed Attributes
   - 6.3 Caching Computed Attributes
   - 6.4 Logging Attribute Access
   - 6.5 Managing Attribute Deletion
   - 6.6 Creating Backwards-Compatible APIs
7. Overriding Properties in Sub-Classes

`property()` In Action

- Managed Attribute Creation
  - Function
  - Decorator
- Read-Only
- Read-Write
- Write-Only

Next: Validating Input Values

Managing Attributes With Python's Property()

6. `property()` In Action
   - 6.1 **Validating Input Values**
   - 6.2 Providing Computed Attributes
   - 6.3 Caching Computed Attributes
   - 6.4 Logging Attribute Access
   - 6.5 Managing Attribute Deletion
   - 6.6 Creating Backwards-Compatible APIs
7. Overriding Properties in Sub-Classes

Validating Input Values

- Building Managed Attributes
- Validation Of Input Data Before
  - Storage
  - Acceptance
- Required When Accepting Data From
  - Users
  - Untrusted Information Sources

Validated `Point` Example

- `.x` and `.y` Must Be Valid Numbers

`raise ... from None`

`raise ValueError()`

```
Traceback (most recent call last):
  File "/Users/darren/python/test.py", line 14, in radius
    self._radius = float(value)
ValueError: could not convert string to float: 'hello'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/darren/python/test.py", line 4, in __init__
    self.radius = radius
  File "/Users/darren/python/test.py", line 17, in radius
    raise ValueError('"x" must be a number')
ValueError: "x" must be a number
```

`raise ValueError() from None`

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/darren/python/test.py", line 4, in __init__
    self.radius = radius
  File "/Users/darren/python/test.py", line 17, in radius
    raise ValueError('"x" must be a number') from None
ValueError: "x" must be a number
```

Python Raise Documentation

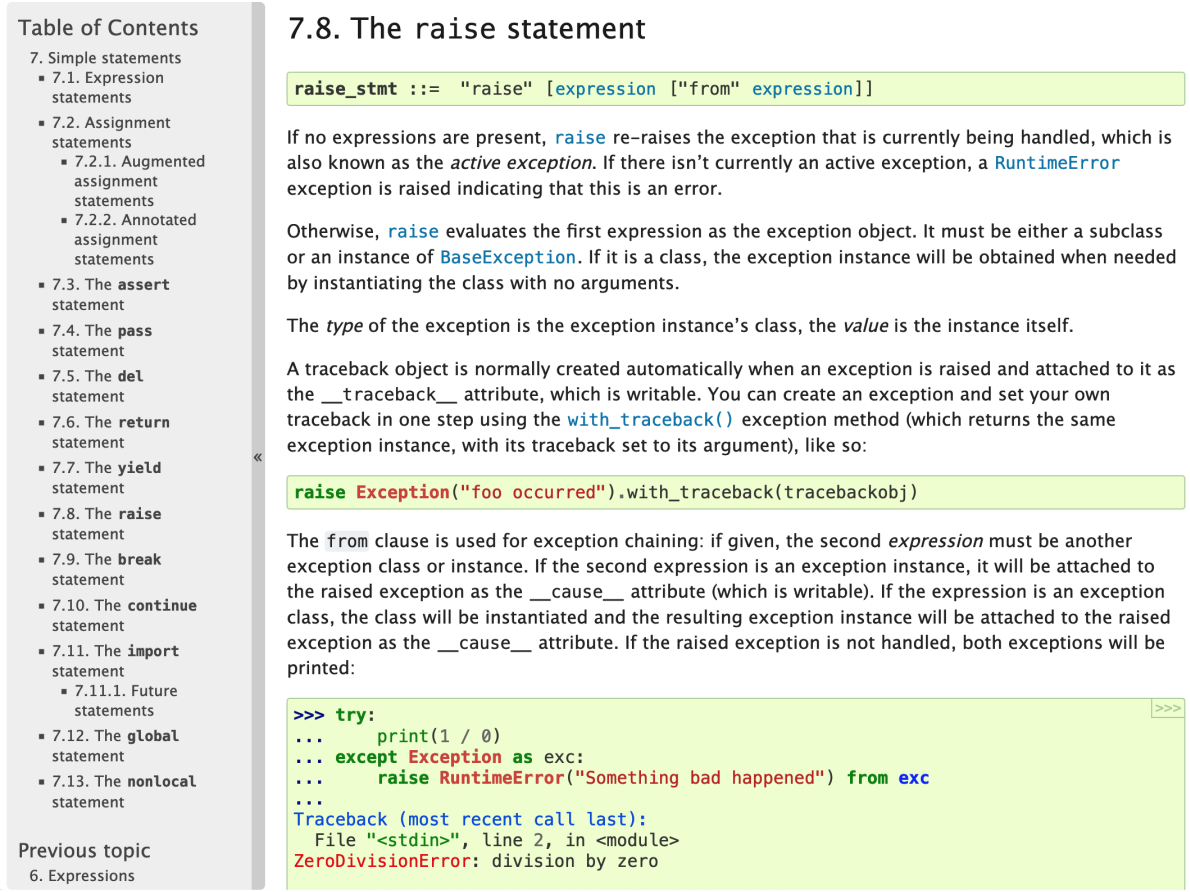


Table of Contents

7. Simple statements
   - 7.1. Expression statements
   - 7.2. Assignment statements
     - 7.2.1. Augmented assignment statements
     - 7.2.2. Annotated assignment statements
   - 7.3. The **assert** statement
   - 7.4. The **pass** statement
   - 7.5. The **del** statement
   - 7.6. The **return** statement
   - 7.7. The **yield** statement
   - 7.8. The **raise** statement
   - 7.9. The **break** statement
   - 7.10. The **continue** statement
   - 7.11. The **import** statement
     - 7.11.1. Future statements
   - 7.12. The **global** statement
   - 7.13. The **nonlocal** statement

Previous topic

6. Expressions

7.8. The raise statement

```
raise_stmt ::=  "raise" [expression ["from" expression]]
```

If no expressions are present, `raise` re-raises the exception that is currently being handled, which is also known as the *active exception*. If there isn't currently an active exception, a `RuntimeError` exception is raised indicating that this is an error.

Otherwise, `raise` evaluates the first expression as the exception object. It must be either a subclass or an instance of `BaseException`. If it is a class, the exception instance will be obtained when needed by instantiating the class with no arguments.

The *type* of the exception is the exception instance's class, the *value* is the instance itself.

A traceback object is normally created automatically when an exception is raised and attached to it as the `__traceback__` attribute, which is writable. You can create an exception and set your own traceback in one step using the `with_traceback()` exception method (which returns the same exception instance, with its traceback set to its argument), like so:

```
raise Exception("foo occurred").with_traceback(tracebackobj)
```

The `from` clause is used for exception chaining: if given, the second *expression* must be another exception class or instance. If the second expression is an exception instance, it will be attached to the raised exception as the `__cause__` attribute (which is writable). If the expression is an exception class, the class will be instantiated and the resulting exception instance will be attached to the raised exception as the `__cause__` attribute. If the raised exception is not handled, both exceptions will be printed:

```
>>> try:
...     print(1 / 0)
... except Exception as exc:
...     raise RuntimeError("Something bad happened") from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

https://docs.python.org/3/reference/simple_stmts.html#the-raise-statement

`property()` Can Lead to Repetition

- Repetition Breaks DRY Principles
- Refactor to Avoid This
- Abstract Out Repetitive Logic Using Descriptors

Consider a Descriptor When Repetition Occurs

Next: Providing Computed Attributes

Managing Attributes With Python's Property()

6. `property()` In Action
   - 6.1 Validating Input Values
   - 6.2 **Providing Computed Attributes**
   - 6.3 Caching Computed Attributes
   - 6.4 Logging Attribute Access
   - 6.5 Managing Attribute Deletion
   - 6.6 Creating Backwards-Compatible APIs
7. Overriding Properties in Sub-Classes

Providing Computed Attributes

- Builds Value Dynamically
- Known as a Computed Attribute
- Appear Like Eager Attributes, But Are Lazy

Eager vs. Lazy Attributes

- Eager Attributes Optimize Computation Cost
- Lazy Attributes Postpone Computation Until Needed

`Rectangle` Class With Computed Attribute

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @property
    def area(self):
        return self.width * self.height
```

`Product` Class With Auto-Formatted Attribute

```
class Product:
    def __init__(self, name, price):
        self._name = name
        self._price = float(price)

    @property
    def price(self):
        return f"${self._price:,.2f}"
```

`Point` Class

- `.x` and `.y` Cartesian Coordinates
- Polar Coordinates Also Needed
  - Distance To Origin
  - Angle from Horizontal Axis

Next: Caching Computed Attributes

Managing Attributes With Python's Property()

6. `property()` In Action
   - 6.1 Validating Input Values
   - 6.2 Providing Computed Attributes
   - 6.3 **Caching Computed Attributes**
   - 6.4 Logging Attribute Access
   - 6.5 Managing Attribute Deletion
   - 6.6 Creating Backwards-Compatible APIs
7. Overriding Properties in Sub-Classes

Caching Computed Attributes

- Some Computed Attributes Used Frequently
- Repeating Calculation May Be Expensive
- Value Can Be Cached in Non-Public Attribute

Input Data Mutability

- Unchanging Input Value
- Result Will Not Change
- Computation Performed Once

`functools.cached_property()`

- Works As a Decorator
- Transforms Method Into a Cached Property
- Computes Value Once

Next: Logging Attribute Access

Managing Attributes With Python's Property()

6. `property()` In Action
   - 6.1 Validating Input Values
   - 6.2 Providing Computed Attributes
   - 6.3 Caching Computed Attributes
   - 6.4 **Logging Attribute Access**
   - 6.5 Managing Attribute Deletion
   - 6.6 Creating Backwards-Compatible APIs
7. Overriding Properties in Sub-Classes

Logging Attribute Access

- `logging` From the Standard Library
- Watch Code
- Generate Information On How Code Works

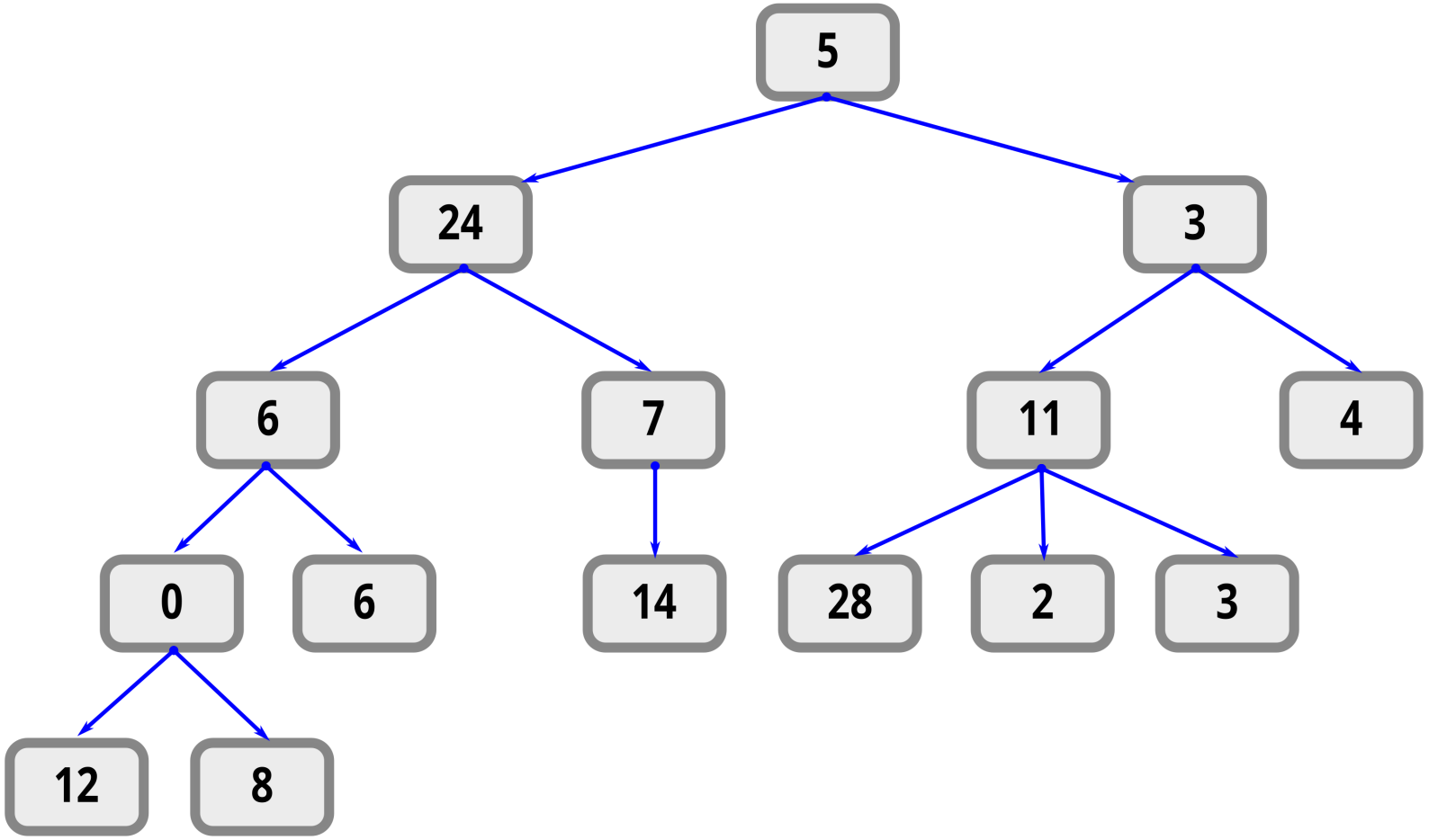
Next: Managing Attribute Deletion

Managing Attributes With Python's Property()

6. `property()` In Action
   - 6.1 Validating Input Values
   - 6.2 Providing Computed Attributes
   - 6.3 Caching Computed Attributes
   - 6.4 Logging Attribute Access
   - 6.5 **Managing Attribute Deletion**
   - 6.6 Creating Backwards-Compatible APIs
7. Overriding Properties in Sub-Classes

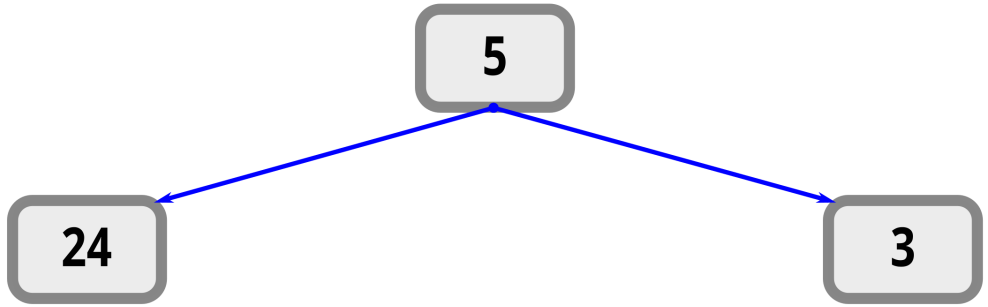
Managing Attribute Deletion

Trees

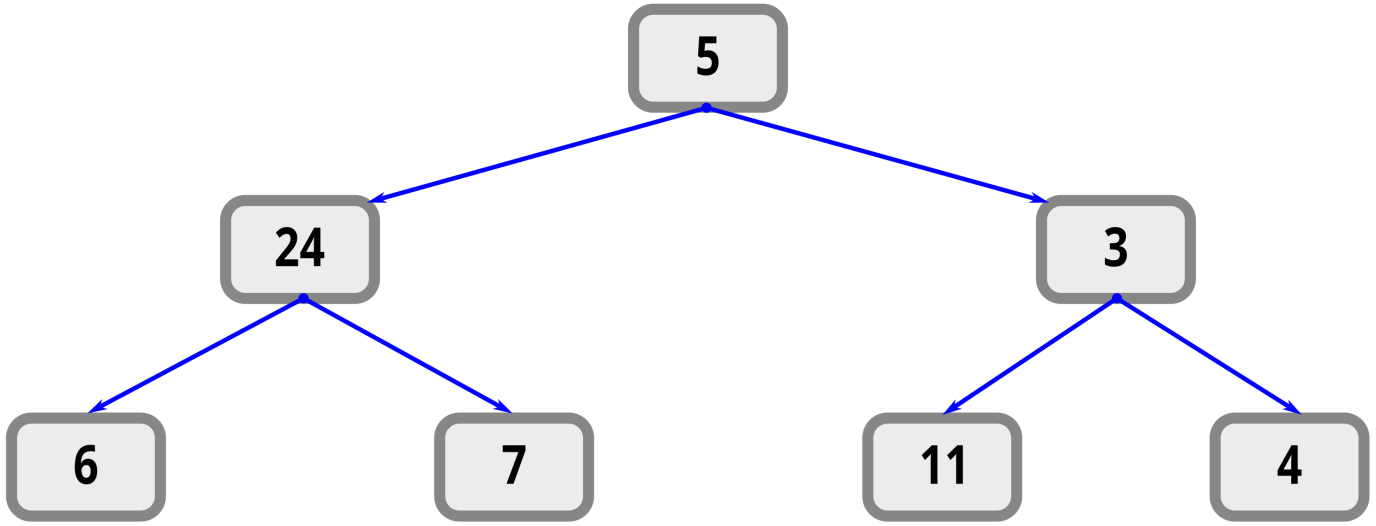


Trees

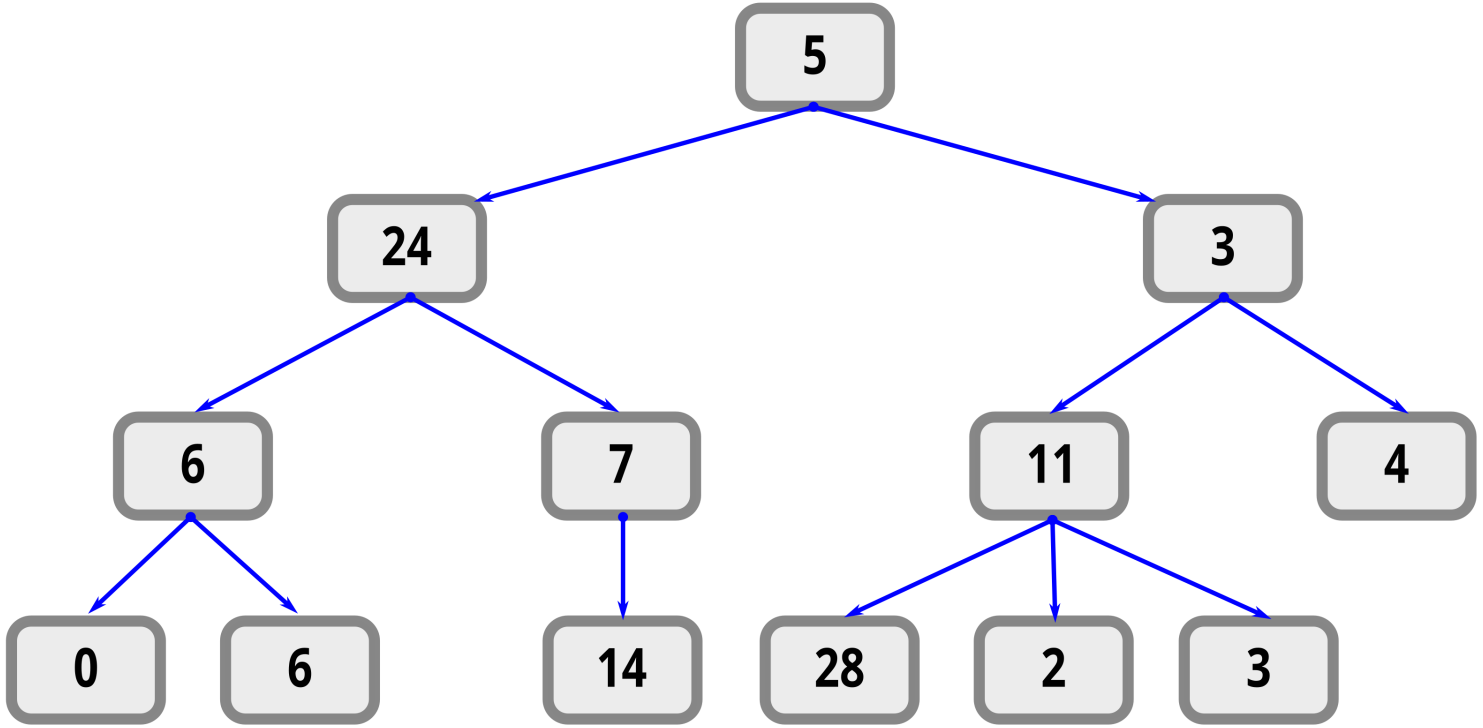
Trees



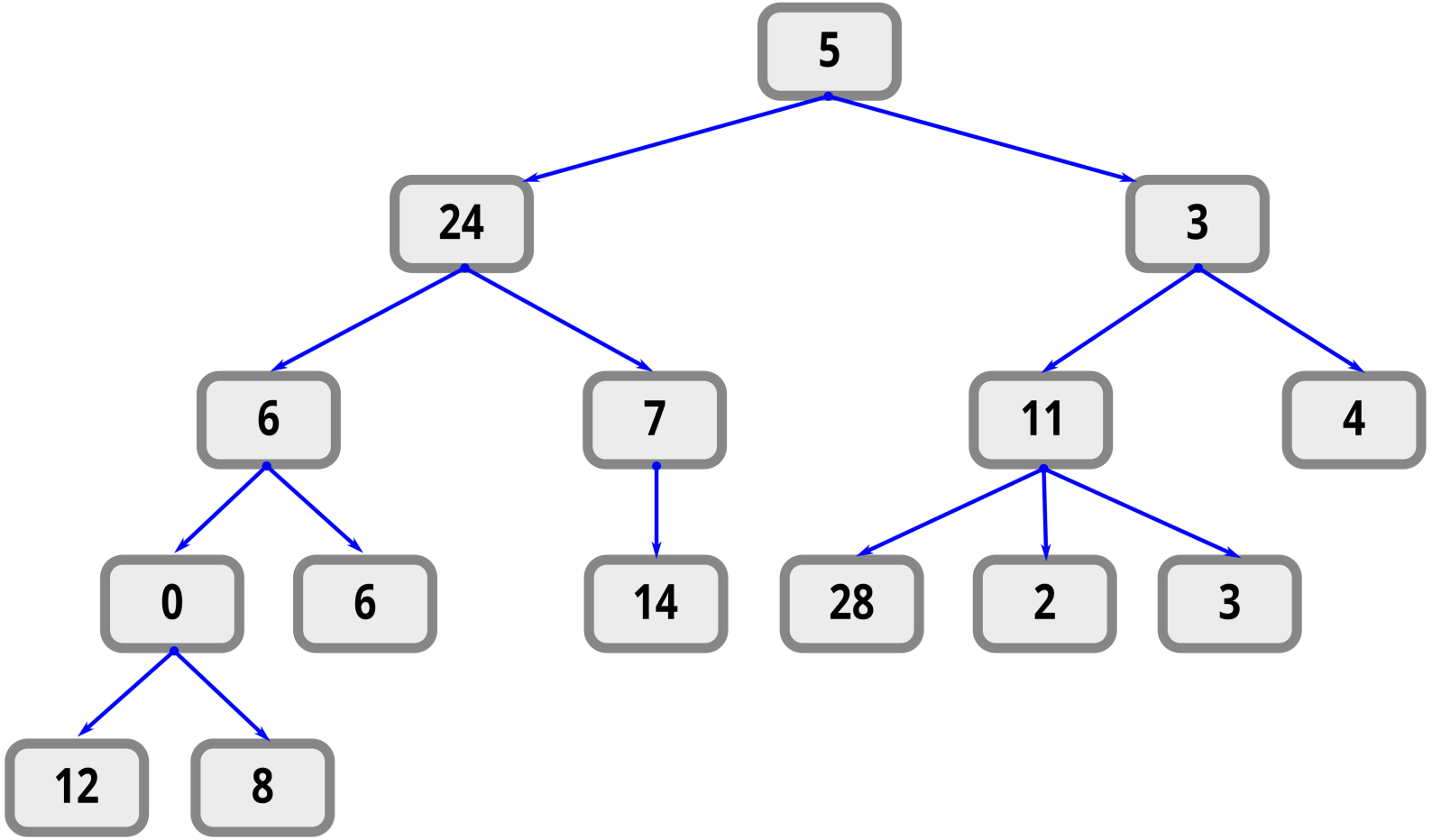
Trees



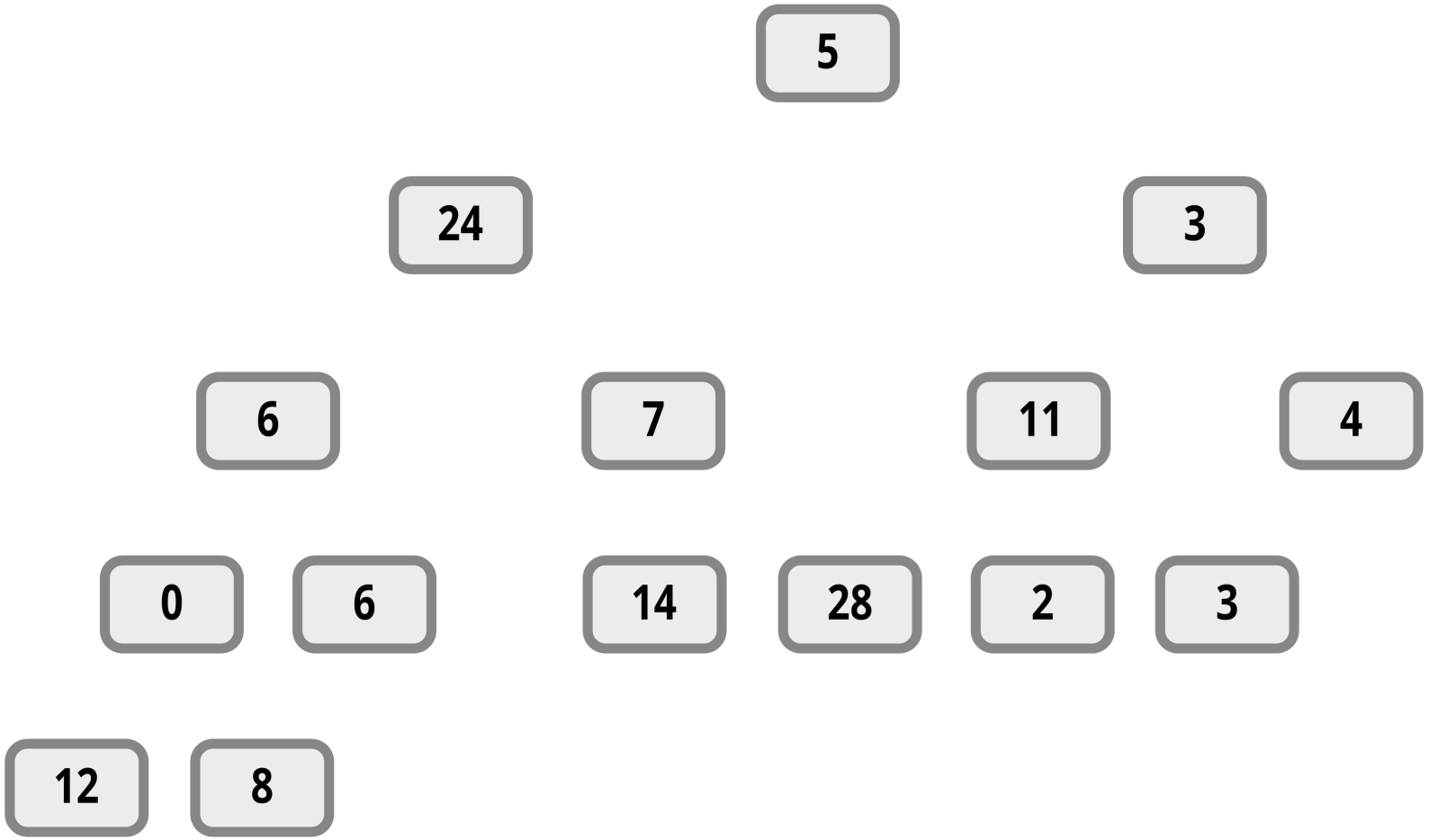
Trees



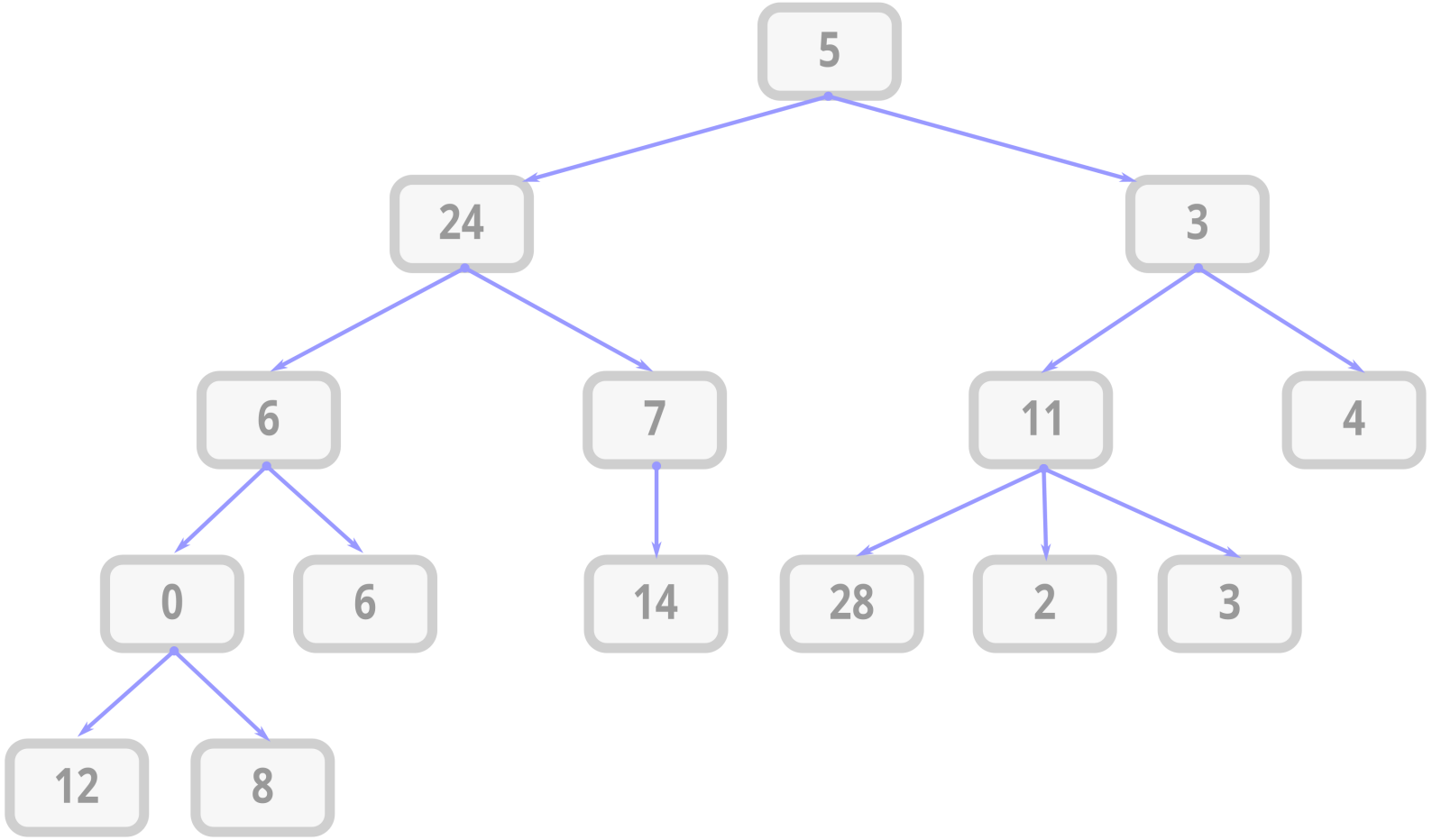
Trees



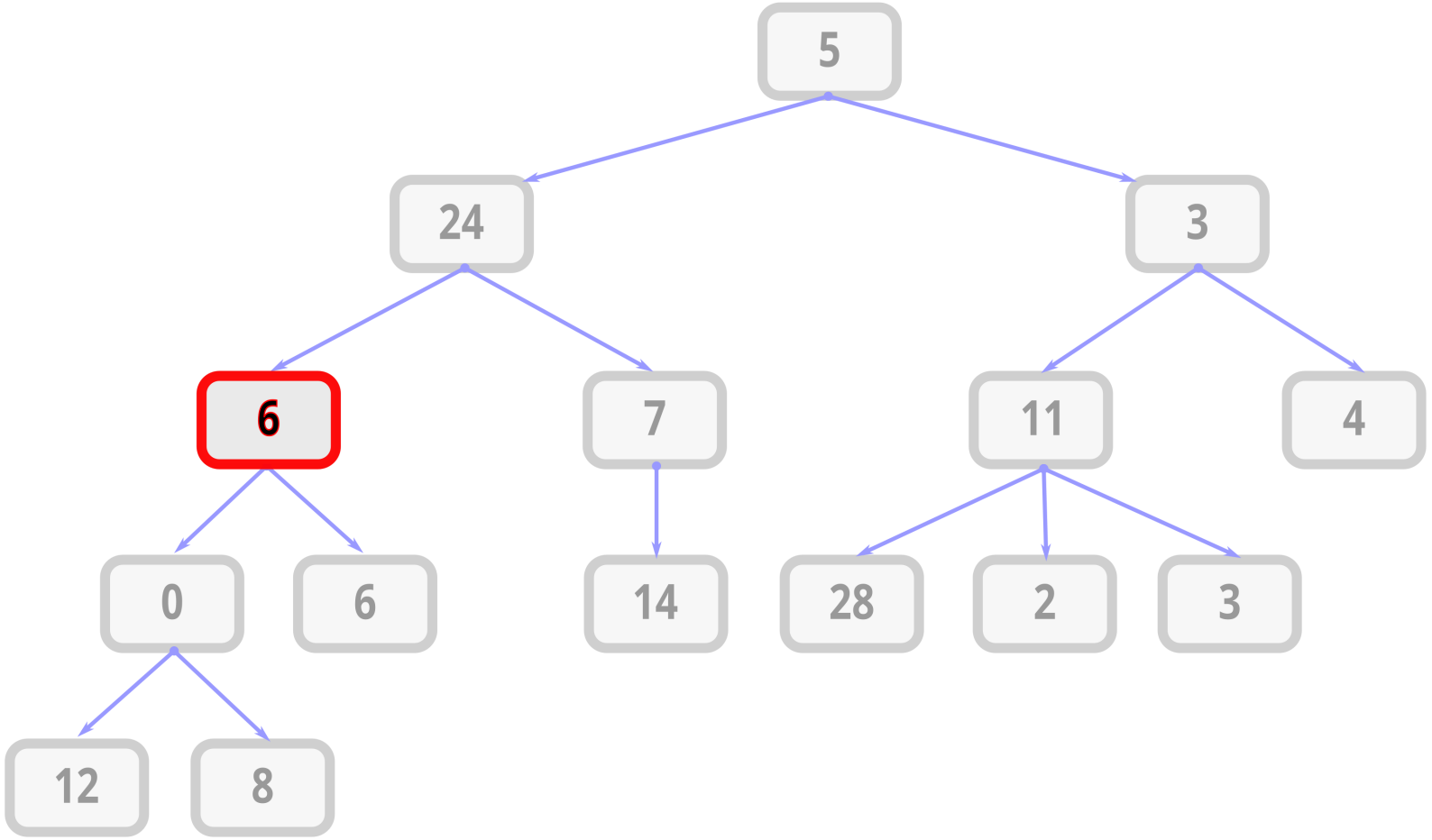
Trees



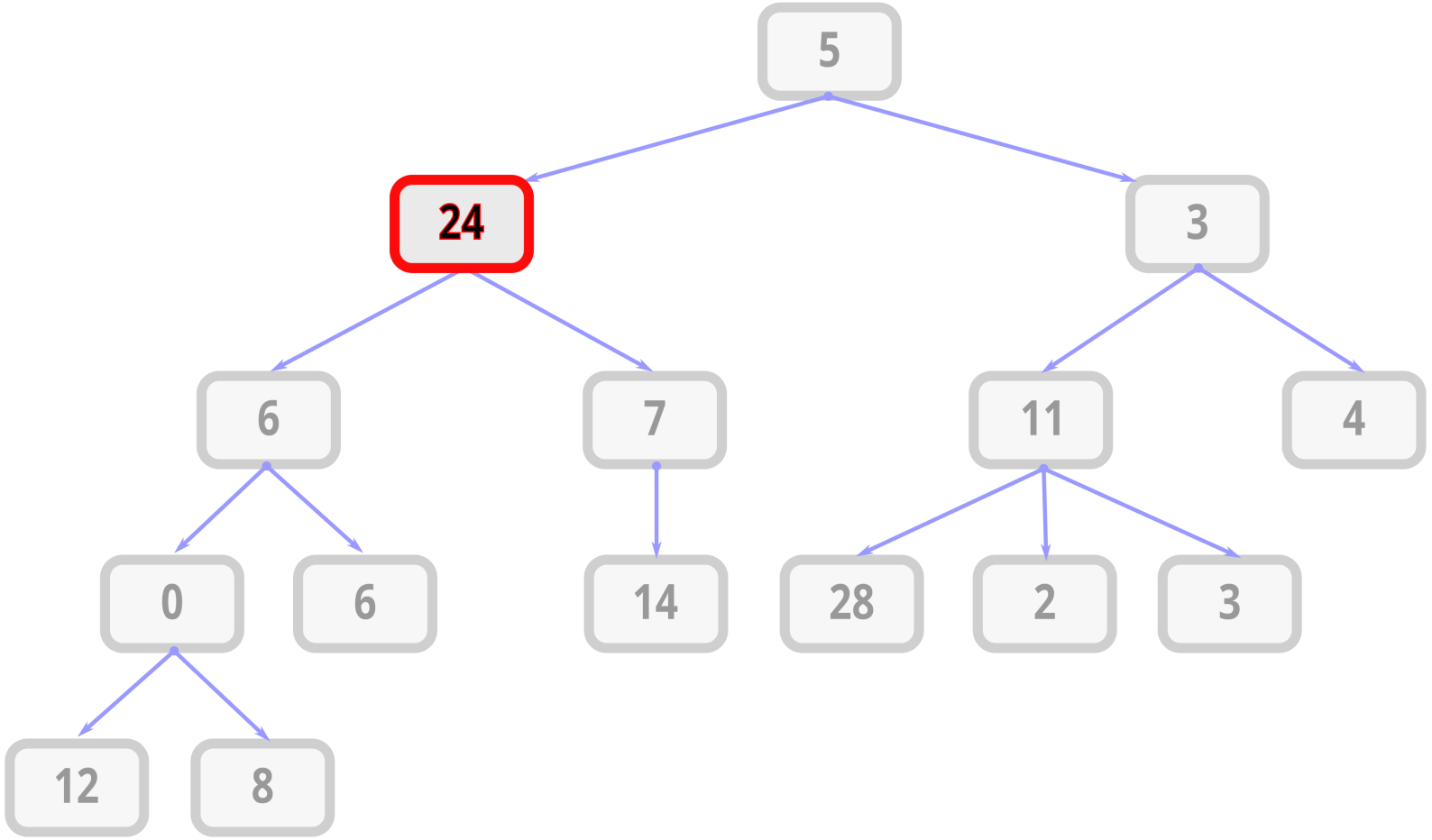
Trees



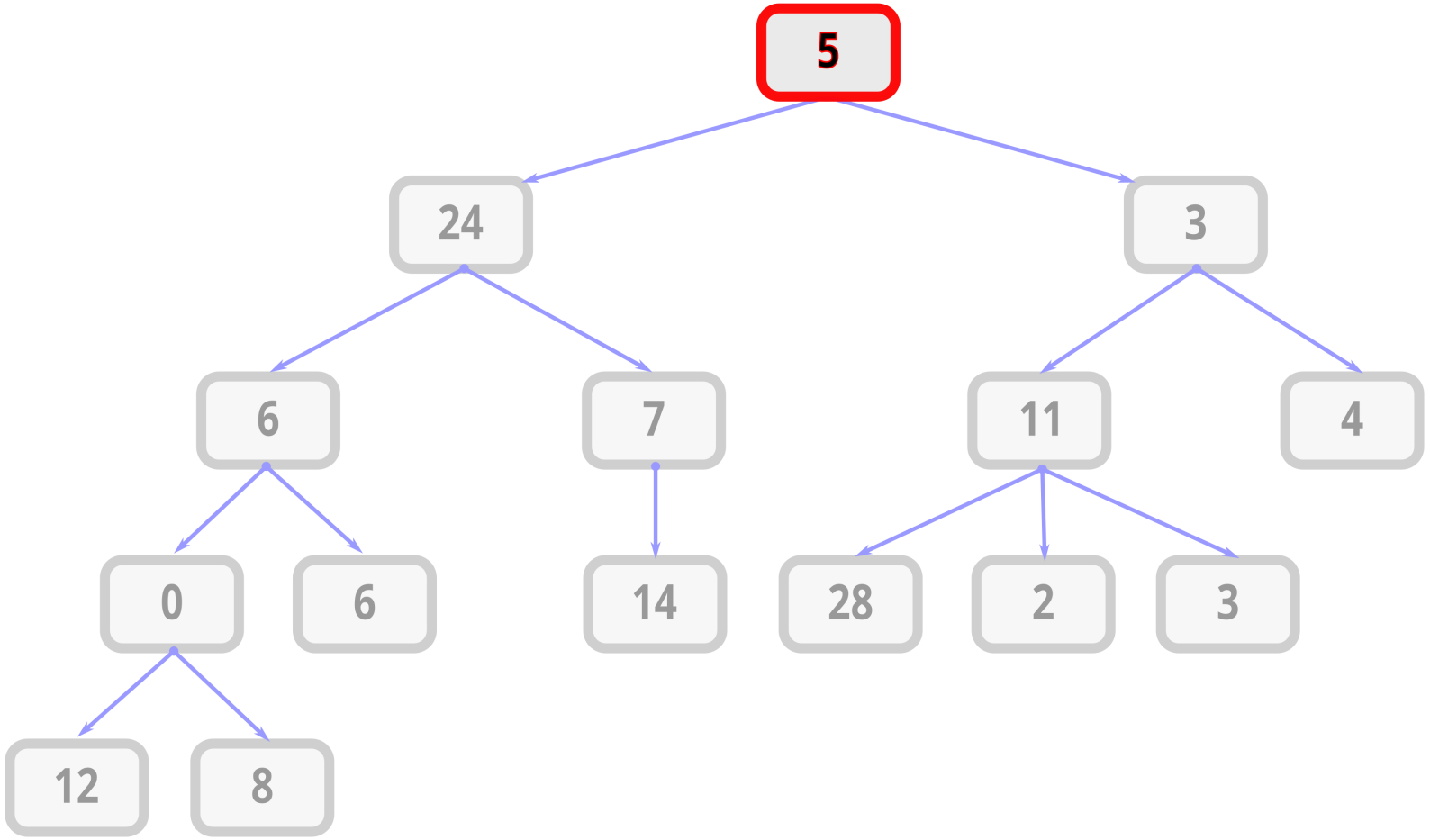
Trees



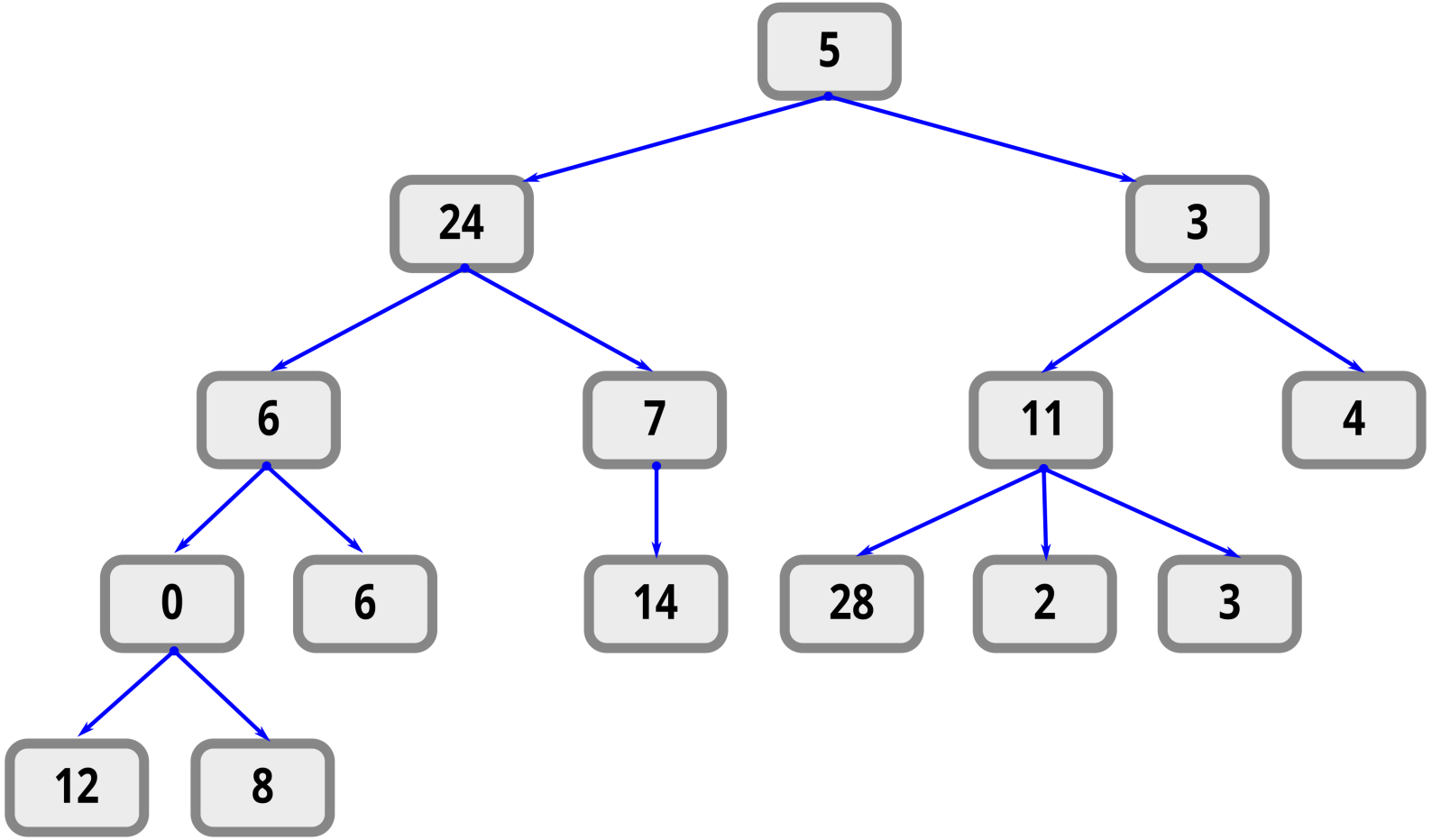
Trees



Trees



Trees



Next: Creating Backwards-Compatible APIs

Managing Attributes With Python's Property()

6. `property()` In Action
   - 6.1 Validating Input Values
   - 6.2 Providing Computed Attributes
   - 6.3 Caching Computed Attributes
   - 6.4 Logging Attribute Access
   - 6.5 Managing Attribute Deletion
   - 6.6 **Creating Backwards-Compatible APIs**
7. Overriding Properties in Sub-Classes

Creating Backwards-Compatible APIs

- `property()` Turns Method Calls Into Attribute Lookups
- Creates Clean, Pythonic APIs
- Attributes Exposed Publicly Without Getters and Setters
- If Modification of an Attribute Needed, Use `property()`
- Processing Possible Without Public API Modification

`Currency` Class

Keep Your Class APIs Stable

Next: Overriding Properties in Sub-Classes

Managing Attributes With Python's Property()

1. Managing Attributes in Your Classes
2. Getting Started With `property()`
3. Providing Read-Only Attributes
4. Providing Read-Write Attributes
5. Providing Write-Only Attributes
6. `property()` In Action
7. **Overriding Properties in Sub-Classes**

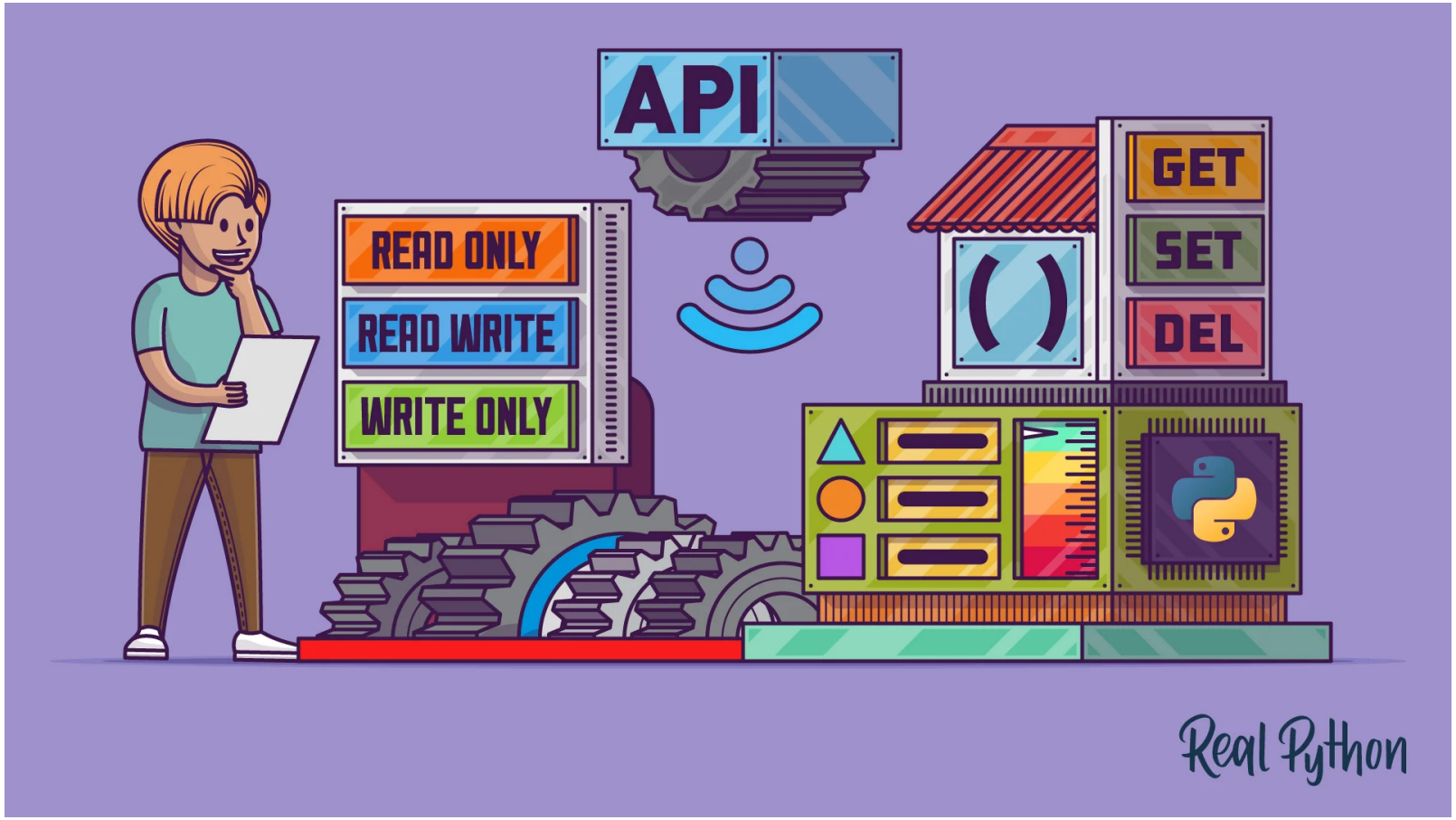
Overriding Properties in Sub-Classes

- Subclassing For Customization
- Partial Override Will Lose Non-Overridden Functionality

`Employee` Class

Next: Summary

Managing Attributes With Python's Property() : Summary



Summary

- Create Managed Attributes With Python's `property()`
- Perform Lazy Attribute Evaluation
- Provide Computed Attributes
- Avoid Setter and Getter Methods
- Create Read-Only, Read-Write and Write-Only Attributes
- Create Consistent and Backward-Compatible APIs

Summary