tcod-ec

Release 2.2.1

Kyle Benesch

Apr 01, 2024

CONTENTS:

CHAPTER

ONE

ABOUT

Entity/Component containers for implementing composition over inheritance.

CHAPTER

TWO

INSTALLATION

Use pip to install this library:

```
pip install tcod-ec
```

If `tcod` is installed and the version is less than `14.0.0` then `import tcod.ec` will fail. Remove or update `tcod` to fix this issue.

CHAPTER

THREE

EXAMPLES

`tcod.ec.ComponentDict` is a container for anonymous components all with a unique class. The key is the class of the component and can only be assigned one instance of that class.

```
>>> import attrs
>>> import tcod.ec

# Anonymous components don't need special treatment.
>>> @attrs.define
... class Position:
...     x: int = 0
...     y: int = 0
>>> @attrs.define
... class Graphic:
...     ch: str = "@"

# ComponentDict stores a single instance for every unique class, in this case: [str,␣
↪Position, Graphic]
>>> entity = tcod.ec.ComponentDict(["Hello world", Position(1, 2), Graphic("!")])
>>> {Position, Graphic} in entity  # Check if an entity has a set of components.
True
>>> entity[str]  # Access components using the class as the key.
'Hello world'
>>> entity[Position].y = 10
>>> entity[Position]
Position(x=1, y=10)
>>> entity[Graphic] = Graphic("?")  # Explicit setting of the component.
>>> entity
ComponentDict(['Hello world', Position(x=1, y=10), Graphic(ch='?')])
>>> entity.set(Graphic("#"))  # Implicit setting.
ComponentDict(['Hello world', Position(x=1, y=10), Graphic(ch='#')])
>>> del entity[Graphic]  # Components can be deleted.
>>> entity
ComponentDict(['Hello world', Position(x=1, y=10)])

# Abstract components can be registered with tcod.ec.abstract_component.
>>> @tcod.ec.abstract_component
... @attrs.define
... class Base:
...     pass
>>> @attrs.define
```

(continues on next page)

(continued from previous page)

```
... class Derived(Base):
...     pass
>>> entity.set(Derived())  # Derived classes may be set implicitly.
ComponentDict(['Hello world', Position(x=1, y=10), Derived()])
>>> entity[Base] = Derived()  # Or explicitly assigned to the abstract key.
>>> Base in entity
True
>>> entity[Base]  # Any derived classes use the base class as the key.
Derived()
>>> entity
ComponentDict(['Hello world', Position(x=1, y=10), Derived()])
```

`tcod.ec.Composite` is a collection of anonymous components. Unlike `ComponentDict` this can store multiple components with the same class. Components can also be accessed using the parent class. This works with multiple inheritance.

While this class looks like an all around upgrade to `ComponentDict` it's far more complex to work with. The ways it mixes class inheritance with composition can lead to anti-patterns if used carelessly. Any class used as a key will return zero, one, or more instances which must be accounted for. If in doubt then the simpler `ComponentDict` should be used instead.

```
>>> @attrs.define
... class Body:
...     name: str
...     hp: int
>>> entity = tcod.ec.Composite([Position(1, 2), Graphic("!"), Body("torso", 10), Body(
↪"head", 5)])
>>> {Position, Graphic, Body} in entity
True
>>> (pos,) = entity[Position]  # Use unpacking logic to verify the number of elements.
>>> pos.y = 10
>>> entity[Position]
[Position(x=1, y=10)]
>>> entity[Graphic] = [Graphic("?")]  # New sequences can be assigned, this deletes all␣
↪previous instances of that key.
>>> entity[Graphic]
[Graphic(ch='?')]
>>> del entity[Graphic]
>>> entity[Graphic]  # Missing keys return an empty sequence instead of KeyError.
()

>>> entity[Body]
[Body(name='torso', hp=10), Body(name='head', hp=5)]
>>> entity.extend([Body("legs", 10), Body("arms", 10)])  # Use append or extend to add␣
↪new instances.
>>> for body in list(entity[Body]):  # Make a copy of the sequence if you intend to␣
↪remove values during iteration.
...     body.hp -= 2
...     if body.name == "torso":
...         entity.remove(body)
>>> entity[Body]
```

(continues on next page)

(continued from previous page)

```
[Body(name='head', hp=3), Body(name='legs', hp=8), Body(name='arms', hp=8)]

# All objects can be accessed at once using `object`.
>>> entity[object]
[Position(x=1, y=10), Body(name='head', hp=3), Body(name='legs', hp=8), Body(name='arms',
↪ hp=8)]
>>> entity[object] = ("Hello", "world")
>>> entity
Composite(['Hello', 'world'])
```

CHAPTER

FOUR

API REFERENCE

Entity/Component containers for implementing composition over inheritance.

Unlike with ECS, these containers are standalone. This makes them simpler to use but they have fewer features.

class tcod.ec.**ComponentDict**(*components=()*, *observers=None*)

Bases: MutableMapping[Type[Any], Any]

A dictionary of component instances, addressed with their class as the key.

This class implements the idea of a `Dict[Type[T], T]` type-hint. This allows adding data and behavior to an object without needing to define which classes the object holds ahead of time.

For equality and hashing ComponentDict's use identity like a standard `object` rather than like a `dict`. This means two different ComponentDict's are considered unique even if they have the same exact components.

```
>>> import attrs
>>> from tcod.ec import ComponentDict
>>> @attrs.define
... class Position:  # Any normal class works as a component.
...     x: int = 0
...     y: int = 0
>>> entity = ComponentDict([Position()])  # Add Position during initialization.
>>> entity.set(Position())  # Or with ComponentDict.set.
ComponentDict([Position(x=0, y=0)])
>>> entity[Position] = Position()  # Or explicitly by key.
>>> entity[Position]  # Access the instance with the class as the key.
Position(x=0, y=0)
>>> {Position} in entity  # Test if an entity has a set of components.
True
>>> @attrs.define
... class Cursor(Position):  # If you need to store a 2nd Position then a subclass␣
↪can be made.
...     pass
>>> entity[Cursor] = Cursor()
>>> entity
ComponentDict([Position(x=0, y=0), Cursor(x=0, y=0)])
>>> ComponentDict([Position(1, 2), Position(3, 4)])  # The same component always␣
↪overwrites the previous one.
ComponentDict([Position(x=3, y=4)])
```

Observers can be used to track the assignment of components. These work best when the components are immutable and frozen.

```
>>> @attrs.define(frozen=True)
... class Position():
...     x: int = 0
...     y: int = 0
>>> def track_position(entity: ComponentDict, new_pos: Position | None, old_pos:␣
↪Position | None) -> None:
...     print(f"Moved: {old_pos} -> {new_pos}")
>>> entity = ComponentDict(components=[Position()], observers={Position: [track_
↪position]})
Moved: None -> Position(x=0, y=0)
>>> entity
ComponentDict([Position(x=0, y=0)], observers=...)
>>> entity[Position] = Position(1, 2)
Moved: Position(x=0, y=0) -> Position(x=1, y=2)
>>> del entity[Position]
Moved: Position(x=1, y=2) -> None
```

Custom functions can be added to the class variable *global_observers* to trigger side-effects on component assignment. This can be used to register components to a global system, handle save migration, or other effects:

```
>>> @attrs.define(frozen=True)
... class Position():
...     x: int = 0
...     y: int = 0
>>> def print_changes(entity: ComponentDict, kind: Type[Any], value: Any | None,␣
↪old_value: Any | None) -> None:
...     print(f"{kind.__name__}: {old_value} -> {value}")
>>> ComponentDict.global_observers.append(print_changes)
>>> entity = ComponentDict([Position()])
Position: None -> Position(x=0, y=0)
>>> entity.set(Position(1, 2))
Position: Position(x=0, y=0) -> Position(x=1, y=2)
ComponentDict([Position(x=1, y=2)])
>>> del entity[Position]
Position: Position(x=1, y=2) -> None
>>> ComponentDict.global_observers.remove(print_changes)
```

Changed in version 2.2: Is now a `collections.abc.MutableMapping` and has all of the relevant methods such as `.values()`.

Parameters

- **components** (*Iterable[object]*) –
- **observers** (*dict[type[Any], list[_ComponentDictObserver[Any]]] | None*) –

__contains__(*keys*)

Return true if the types of component exist in this entity. Takes a single type or an iterable of types.

Changed in version 1.2: Now supports checking multiple types at once.

Parameters
keys (*type[object] | Iterable[type[object]]*) –

Return type
bool

__delitem__(*key*)

Delete a component.

Parameters
key (*type[object]*) –

Return type
None

__getitem__(*key*)

Return a component of type, raises KeyError if it doesn't exist.

Parameters
key (*type[T]*) –

Return type
T

__getstate__()

Pickle this instance. Any subclass slots and dict attributes will also be saved.

Return type
dict[str, *Any*]

__iter__()

Iterate over the keys of this container.

Return type
Iterator[type[*Any*]]

__len__()

Return the number of components contained in this object.

Return type
int

__missing__(*key*)

Raise KeyError with the missing key. Called when a key is missing.

Example:

```
class DefaultComponentDict(ComponentDict):
    __slots__ = ()

    def __missing__(self, key: Type[T]) -> T:
        """Create default components for missing keys by calling `key` without␣
↪parameters."""
        self[key] = value = key()
        return value
```

Parameters
key (*type[T]*) –

Return type
T

__setitem__(*key*, *value*)

Set or replace a component.

Parameters

- **key** (*type[T]*) –
- **value** (*T*) –

Return type
None

__setstate__(*state*)

Unpickle instances from 1.0 or later, or complex subclasses from 2.0 or later.

Component classes can change between picking and unpickling, and component order should be preserved.

Side-effects are triggered. The unpickled object acts as if its components are newly assigned to it when observed.

Parameters
state (*Any | dict[str, Any]*) –

Return type
None

global_observers: ClassVar[list[Callable[[*ComponentDict*, Type[Any], Any | None, Any | None], None]]] = []

A class variable list of functions to call with component changes.

Unpickled and copied objects are observed as if their components are newly created.

These work best with frozen immutable types as components if you want to observe all value changes.

This can be used to improvise the "systems" of ECS. Observers can collect types of components in a global registry for example.

Example:

```
from typing import Any, Type
import tcod.ec

def my_observer(entity: ComponentDict, kind: Type[Any], value: Any | None, old_
↪value: Any | None) -> None:
    """Print observed changes in components."""
    print(f"{entity=}, {kind=}, {value=}, {old_value=}")

tcod.ec.ComponentDict.global_observers.append(my_observer)
```

New in version 2.0.

Warning: Components in a garbage collected entity are not observed as being deleted. Use *clear* when you are finished with an entity and want its components observed as being deleted.

observers: dict[type[Any], list[Callable[[*ComponentDict*, Any | None, Any | None], None]]]

A dictionary of a list of component observers.

New in version 2.2.

set(**components*)

Assign or replace the components of this entity and return self.

Changed in version 1.1: Now returns self.

Parameters
components (*object*) –

Return type
Self

class tcod.ec.**Composite**(*components=()*)

Bases: object

A collection of multiple components organized by their inheritance tree.

Allows multiple components for the same class and allows accessing components using a shared parent class.

The order of components is preserved.

```
>>> import attrs
>>> from tcod.ec import Composite
>>> @attrs.define
... class AreaOfEffect:
...     range: int
>>> @attrs.define
... class Circle(AreaOfEffect):
...     pass
>>> @attrs.define
... class Square(AreaOfEffect):
...     pass
>>> spell = Composite([50, "fire", "damage", Circle(5)])
>>> spell[int]
[50]
>>> spell[str]
['fire', 'damage']
>>> spell[AreaOfEffect]
[Circle(range=5)]
>>> spell[Circle]
[Circle(range=5)]
>>> spell[Square]
()
>>> spell[object]
[50, 'fire', 'damage', Circle(range=5)]
>>> spell.remove('damage')
>>> spell.add("effect")
>>> spell
Composite([50, 'fire', Circle(range=5), 'effect'])
>>> spell[int] = (20,)
>>> spell[int]
[20]
>>> spell
Composite(['fire', Circle(range=5), 'effect', 20])
>>> spell[AreaOfEffect] = (Square(3),)
>>> spell
Composite(['fire', 'effect', 20, Square(range=3)])
```

New in version 2.1.

Parameters
components (*Iterable[object]*) –

__contains__(*keys*)

Return true if all types or sub-types of *keys* exist in this entity.

Takes a single type or an iterable of types.

Parameters
keys (*type[object] | Iterable[type[object]]*) –

Return type
bool

__delitem__(*key*)

Remove all instances of *key* if they exist.

Parameters
key (*type[object]*) –

Return type
None

__getitem__(*key*)

Return a sequence of all instances of *key*.

If no instances of *key* are stored then return an empty sequence.

The actual list returned is internal and should not be saved. Copy the value with `tuple` or `list` if you intend to store the sequence. Do not modify the sequence.

Parameters
key (*type[T]*) –

Return type
Sequence[*T*]

__getstate__()

Pickle this instance. Any subclass slots and dict attributes will also be saved.

Return type
dict[str, *Any*]

__setitem__(*key*, *values*)

Replace all instances of *key* with the instances of *values*.

Parameters

- **key** (*type[T]*) –
- **values** (*Iterable[T]*) –

Return type
None

__setstate__(*state*)

Unpickle instances of this object.

Any class changes in pickled components will be reflected correctly.

Parameters
state (*dict[str, Any]*) –

Return type
None

add(*component*)

Add a component to this container.

Parameters
component (*object*) –

Return type
None

clear()

Clear all components from this container.

Return type
None

extend(*components*)

Add multiple components to this container.

Parameters
components (*Iterable[object]*) –

Return type
None

remove(*component*)

Remove a component from this container.

Will raise ValueError if the component was not present.

Parameters
component (*object*) –

Return type
None

tcod.ec.**abstract_component**(*cls*)

Register class *cls* as an abstract component and return it.

Subclasses of this *cls* will now use *cls* as the key when being accessed in *ComponentDict*. This means that ComponentDict can only hold one unique instance of this subclass.

Example:

```
>>> from tcod.ec import ComponentDict, abstract_component
>>> from attrs import define
>>> @abstract_component
... @define
... class Base:
...     pass
>>> @define
... class Derived(Base):
...     pass
>>> entity = ComponentDict([Derived()])
>>> entity.set(Derived())
ComponentDict([Derived()])
>>> entity[Base] = Derived()  # Note there can only be one instance assigned to an␣
↪abstract component class.
>>> Base in entity
True
```

(continues on next page)

(continued from previous page)

```
>>> entity[Base]  # Access Base or Derived with the abstract component class.
Derived()
>>> entity[Base] = Base()
>>> entity[Base]
Base()
>>> entity.set(Derived())
ComponentDict([Derived()])
>>> entity[Base]
Derived()
```

Parameters
cls (*type[T]*) –

Return type
type[*T*]

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

Symbols

A

C

E

G

M

O

R

S

T