Python Utilities Documentation

Release 0.0.0

Marcus von Appen

Sep 18, 2017

Contents

Python utility classes. Use them as you wish.

CHAPTER 1

Contents

array - Converting sequences

This module provides various functions and classes to access sequences and buffer-style objects in different ways. It also provides conversion routines to improve the interoperability of sequences with `ctypes` data types.

Providing read-write access for sequential data

Two classes allow you to access sequential data in different ways. The *`CTypesView`* provides byte-wise access to iterable objects and allows you to convert the object representation to matching byte-widths for `ctypes` or other modules.

Depending on the the underlying object and the chosen size of each particular item of the object, the *`CTypesView`* allows you to operate directly on different representations of the object's contents.

```
>>> text = bytearray("Hello, I am a simple ASCII string!")
>>> ctview = CTypesView(text, itemsize=1)
>>> ctview.view[0] = 0x61
>>> print(text)
aello, I am a simple ASCII string!"
>>> ctview.to_uint16()[3] = 0x6554
>>> print(text)
aello,Te am a simple ASCII string!"
```

The snippet above provides a single-byte sized view on a `bytearray()` object. Afterwards, the first item of the view is changed, which causes a change on the `bytearray()`, on the first item as well, since both, the *`CTypesView`* and the `bytearray()` provide a byte-wise access to the contents.

By using *`CTypesView.to_uint16()`*, we change the access representation to a 2-byte unsigned integer `ctypes` pointer and change the fourth 2-byte value, *I* to something else.

```
>>> text = bytearray("Hello, I am a simple ASCII string!")
>>> ctview = CTypesView(text, itemsize=2)
```

```
>>> ctview.view[0] = 0x61
>>> print(text)
aello, I am a simple ASCII string!"
>>> ctview.to_uint16()[3] = 0x6554
>>> print(text)
aello,Te am a simple ASCII string!"
```

If the encapsuled object does not provide a (writeable) `buffer()` interface, but is iterable, the *CTypesView* will create an internal copy of the object data using Python's *array* module and perform all operations on that copy.

```
>>> mylist = [18, 52, 86, 120, 154, 188, 222, 240]
>>> ctview = CTypesView(mylist, itemsize=1, docopy=True)
>>> print(ctview.object)
array('B', [18, 52, 86, 120, 154, 188, 222, 240])
>>> ctview.view[3] = 0xFF
>>> print(mylist)
[18, 52, 86, 120, 154, 188, 222, 240]
>>> print(ctview.object)
array('B', [18, 52, 86, 255, 154, 188, 222, 240])
```

As for directly accessible objects, you can define your own itemsize to be used. If the iterable does not provide a direct byte access to their contents, this won't have any effect except for resizing the item widths.

```
>>> mylist = [18, 52, 86, 120, 154, 188, 222, 240]
>>> ctview = CTypesView(mylist, itemsize=4, docopy=True)
>>> print(ctview.object)
array('I', [18L, 52L, 86L, 120L, 154L, 188L, 222L, 240L])
```

Accessing data over multiple dimensions

The second class, *MemoryView* provides an interface to access data over multiple dimensions. You can layout and access a simple byte stream over e.g. two or more axes, providing a greater flexibility for functional operations and complex data.

Let's assume, we are reading image data from a file stream into some buffer object and want to access and manipulate the image data. Images feature two axes, one being the width, the other being the height, defining a rectangular graphics area.

When we read all data from the file, we have an one-dimensional view of the image graphics. The *MemoryView* allows us to define a two-dimensional view over the image graphics, so that we can operate on both, rows and columns of the image.

```
>>> imagedata = bytearray("some 1-byte graphics data")
>>> view = MemoryView(imagedata, 1, (5, 5))
>>> print(view)
[[s, o, m, e,  ], [1, -, b, y, t], [e,  , g, r, a], [p, h, i, c, s], [ , d, a, t, a]]
>>> for row in view:
...     print(row)
...
[s, o, m, e,  ]
[1, -, b, y, t]
[e,  , g, r, a]
[p, h, i, c, s]
[ , d, a, t, a]
>>> for row in view:
...     row[1] = "X"
```

```
...     print row
...
[s, X, m, e,  ]
[1, X, b, y, t]
[e, X, g, r, a]
[p, X, i, c, s]
[ , X, a, t, a]
>>> print(imagedata)
sXme 1XbyteXgrapXics Xata
```

On accessing a particular dimension of a *MemoryView*, a new *MemoryView* is created, if it does not access a single element.

```
>>> firstrow = view[0]
>>> type(firstrow)
<class 'mule.array.MemoryView'>
>>> type(firstrow[0])
<type 'bytearray'>
```

A *MemoryView* features, similar to Python's builtin memoryview, dimensions and strides, accessible via the *MemoryView.ndim* and *MemoryView.strides* attributes.

```
>>> view.ndim
2
>>> view.strides
(5, 5)
```

The *MemoryView.strides*, which have to be passed on creating a new *MemoryView*, define the layout of the data over different dimensions. In the example above, we created a 5x5 two-dimensional view to the image graphics.

```
>>> twobytes = MemoryView(imagedata, 2, (5, 1))
>>> print(twobytes)
[[sX, me,  1, Xb, yt], [eX, gr, ap, Xi, cs]]
```

API

class array.**CTypesView**(*obj : iterable*[, *itemsize=1*[, *docopy=False*[, *objsize=None*]]])

A proxy class for byte-wise accessible data types to be used in ctypes bindings. The CTypesView provides a read-write access to arbitrary objects that are iterable.

In case the object does not provide a buffer() interface for direct access, the CTypesView can copy the object's contents into an internal buffer, from which data can be retrieved, once the necessary operations have been performed.

Depending on the item type stored in the iterable object, you might need to provide a certain *itemsize*, which denotes the size per item in bytes. The *objsize* argument might be necessary of iterables, for which len() does not return the correct amount of objects or is not implemented.

bytesize

Returns the length of the encapsuled object in bytes.

is_shared

Indicates, if changes on the CTypesView data effect the encapsuled object directly. if not, this means that the object was copied internally and needs to be updated by the user code outside of the CTypesView.

object

The encapsuled object.

view
Provides a read-write aware view of the encapsuled object data that is suitable for usage from `ctypes`.

to_bytes() → ctypes.POINTER
Returns a byte representation of the encapsuled object. The return value allows a direct read-write access to the object data, if it is not copied. The `ctypes.POINTER()` points to an array of `ctypes.c_ubyte`.

to_uint16() → ctypes.POINTER
Returns a 16-bit representation of the encapsuled object. The return value allows a direct read-write access to the object data, if it is not copied. The `ctypes.POINTER()` points to an array of `ctypes.c_ushort`.

to_uint32() → ctypes.POINTER
Returns a 32-bit representation of the encapsuled object. The return value allows a direct read-write access to the object data, if it is not copied. The `ctypes.POINTER()` points to an array of `ctypes.c_uint`.

to_uint64() → ctypes.POINTER
Returns a 64-bit representation of the encapsuled object. The return value allows a direct read-write access to the object data, if it is not copied. The `ctypes.POINTER()` points to an array of `ctypes.c_ulonglong`.

class `array.`**MemoryView**(*source : object*, *itemsize : int*, *strides : tuple*[, *getfunc=None*[, *setfunc=None*[, *srcsize=None*]]])
The *MemoryView* provides a read-write access to arbitrary data objects, which can be indexed.

itemsize denotes the size of a single item. *strides* defines the dimensions and the length (n items * *itemsize*) for each dimension. *getfunc* and *setfunc* are optional parameters to provide specialised read and write access to the underlying *source*. *srcsize* can be used to provide the correct source size, if `len(source)` does not return the absolute size of the source object in all dimensions.

Note: The MemoryView is a pure Python-based implementation and makes heavy use of recursion for multi-dimensional access. If you aim for speed on accessing a n-dimensional object, you want to consider using a specialised library such as numpy. If you need n-dimensional access support, where such a library is not supported, or if you need to provide access to objects, which do not fulfill the requirements of that particular libray, *MemoryView* can act as solid fallback solution.

---

itemsize
The size of a single item in bytes.

ndim
The number of dimensions of the *MemoryView*.

size
The size in bytes of the underlying source object.

source
The underlying data source.

strides
A tuple defining the length in bytes for accessing all elements in each dimension of the *MemoryView*.

`array.`**to_ctypes**(*dataseq : iterable*, *dtype*[, *mcount=0*]) → array, int
Converts an arbitrary sequence to a ctypes array of the specified *dtype* and returns the ctypes array and amount of items as two-value tuple.

Raises a `TypeError`, if one or more elements in the passed sequence do not match the passed *dtype*.

`array.`**to_list**(*dataseq : iterable*) → list
Converts a ctypes array to a list.

`array.`**`to_tuple`**(*dataseq : iterable*) → tuple
Converts a ctypes array to a tuple.

`array.`**`create_array`**(*obj : object*, *itemsize : int*) → array.array
Creates an `array.array` based copy of the passed object. *itemsize* denotes the size in bytes for a single element within *obj*.

compat - Python compatibility helpers

The *compat* module is for internal purposes of your package or application and should not be used outside of it.

`compat.`**`ISPYTHON2`**
`True`, if executed in a Python 2.x compatible interpreter, `False` otherwise.

`compat.`**`ISPYTHON3`**
`True`, if executed in a Python 3.x compatible interpreter, `False` otherwise.

`compat.`**`long`**(*[x[, base]]*)

Note: Only defined for Python 3.x, for which it is the same as `int()`.

`compat.`**`unichr`**(*i*)

Note: Only defined for Python 3.x, for which it is the same as `chr()`.

`compat.`**`unicode`**(*string[, encoding[, errors]]*)

Note: Only defined for Python 3.x, for which it is the same as `str()`.

`compat.`**`callable`**(*x*) → bool

Note: Only defined for Python 3.x, for which it is the same as `isinstance(x, collections.Callable)`

`compat.`**`byteify`**(*x : string*, *enc : string*) → bytes
Converts a string to a `bytes()` object.

`compat.`**`stringify`**(*x : bytes*, *enc : string*) → string
Converts a `bytes()` to a string object.

`compat.`**`isiterable`**(*x*) → bool
Shortcut for `isinstance(x, collections.Iterable)`.

`compat.`**`platform_is_64bit`**() → bool
Checks, if the interpreter is 64-bit capable.

`@compat.`**`deprecated`**
A simple decorator to mark functions and methods as deprecated. This will print a deprecation message each time the function or method is invoked.

`compat.`**`deprecation`**(*message : string*) → None
Prints a deprecation message using the `warnings.warn()` method.

exception `compat.`**`UnsupportedError`**(*obj : object*[, *msg=None*])
Indicates that a certain class, function or behaviour is not supported in the specific execution environment.

`@compat.`**`experimental`**
A simple decorator to mark functions and methods as experimental. This will print a warning each time the function or method is invoked.

exception `compat.`**`ExperimentalWarning`**(*obj : object*[, *msg=None*])
Indicates that a certain class, function or behaviour is in an experimental state.

dll - DLL loading

The *dll* module is not intended for consumers of your specific application or library. It is a helper module for loading the 3rd party libraries used by your project itself.

class `dll.`**`DLL`**(*libinfo : string*, *libnames : string or dict*[, *path=None*])
A simple wrapper class for loading shared libraries through ctypes.

The *libinfo* argument is a descriptive name of the library, that is recommended to be platform neutral, since it is shown to the user on errors. *libnames* can be a list of shared library names or a dictionary consisting of platform->library name mappings. *path* is the explicit library path to be used, if any. *path* acts as the first location to be used for loading the library, before the standard mechanisms of `ctypes` will be used

`libfile`
Gets the filename of the loaded library.

`bind_function`(*funcname : string*[, *args=None*[, *returns=None*[, *optfunc=None*]]]) → function
Tries to resolve the passed function name and, if found, binds the list of *args*, to its `argtypes` and the *returns* value to its `restype`. If the function is not found, *optfunc* will be used instead, without the assignment of *args* and *returns*.

dojo - Training classes for functions and algorithms

class `dojo.`**`Dojo`**(*algorithms : sequence*, *environ : object*[, *runs=100*])
A simple testing class for competing algorithms. *Dojo* is the base class for concrete implementations to measure the performance of algorithms that shall perform the same task.

`algorithms`
The list of algorithms to compare

`environ`
An arbeitrary object that simulates the execution environment for the algorithms. A copy of it will be passed as first argument to each algorithm.

Note: The copy will be created via `copy.deepcopy()` to ensure that (hopefully) no value of the original environment will be modified by an algorithm.

---

runs
The number of consecutive runs for each algorithm.

train(**args*) → None
Executes the algorithms, comparing their performance.

This has to be implemented by inheriting classes.

class dojo.**TimingDojo**(*algorithms : sequence*, *environ : object*[, *runs=100*])
A *Dojo* implementation measuring the execution time of its algorithms.

train(**args*) → object
Executes the algorithms and compares their run-time performance *Dojo.runs* is used to create a reliable average mean for the execution time and hence should not be chosen too small.

The method will return the best performing algorithm.

class dojo.**FitnessDojo**(*algorithms : sequence*, *environ : object*[, *runs=100*[, *cmpfunc=min*]])
A *Dojo* implementation measuring the fitness of its algorithms. The fitness is determined by the passed *cmpfunc*.

cmpfunc
Comparision function for the fitness measurement. It has to return a single object of a passed iterable and must accept a named argument `key`, since its execution looks like

```
dojo.cmpfunc(result_dict, key=result_dict.get)
```

train(**args*) → object
Executes the algorithms and compares their return value, which *must* be a float. *Dojo.runs* is used to create a reliable average mean for the return value and hence should not be chosen too small.

ebs - A component-based entity system framework

This module loosely follows a component oriented pattern to separate object instances, carried data and processing logic within applications or games. It uses an entity based approach, in which object instances are unique identifiers, while their data is managed within components, which are separately stored. For each individual component type a processing system will take care of all necessary updates on running the application.

Component-based patterns

Component-based means that - instead of a traditional OOP approach - object information are split up into separate data bags for reusability and that those data bags are separated from any application logic.

Behavioural design

Imagine a car game class in traditional OOP, which might look like

```
class Car:
    def __init__(self):
        self.color = "red"
        self.position = 0, 0
        self.velocity = 0, 0
        self.sprite = get_some_car_image()
        ...
    def drive(self, timedelta):
```

```
        self.position[0] = self.velocity[0] * timedelta
        self.position[1] = self.velocity[1] * timedelta
        ...
    def stop(self):
        self.velocity = 0, 0
        ...
    def render(self, screen):
        screen.display(self.sprite)

mycar = new Car()
mycar.color = "green"
mycar.velocity = 10, 0
```

The car features information stored in attributes (`color`, `position`, ...) and behaviour (application logic, `drive()`, `stop()` ...).

A component-based approach aims to split and reduce the car to a set of information and external systems providing the application logic.

```
class Car:
    def __init__(self):
        self.color = "red"
        self.position = 0, 0
        self.velocity = 0, 0
        self.sprite = get_some_car_image()

class CarMovement:
    def drive(self, car, timedelta):
        car.position[0] = car.velocity[0] * timedelta
        car.position[1] = car.velocity[1] * timedelta
        ...
    def stop(self):
        car.velocity = 0, 0

class CarRenderer:
    def render(self, car, screen):
        screen.display(car.sprite)
```

At this point of time, there is no notable difference between both approaches, except that the latter one adds additional overhead.

The benefit comes in, when you

- use subclassing in your OOP design
- want to change behavioural patterns on a global scale or based on states
- want to refactor code logic in central locations
- want to cascade application behaviours

The initial `Car` class from above defines, how it should be displayed on the screen. If you now want to add a feature for rescaling the screen size after the user activates the magnifier mode, you need to refactor the `Car` and all other classes that render things on the screen, have to consider all subclasses that override the method and so on. Refactoring the `CarRenderer` code by adding a check for the magnifier mode sounds quite simple in contrast to that, not?

The same applies to the movement logic - inverting the movement logic requires you to refactor all your classes instead of a single piece of application code.

Information design

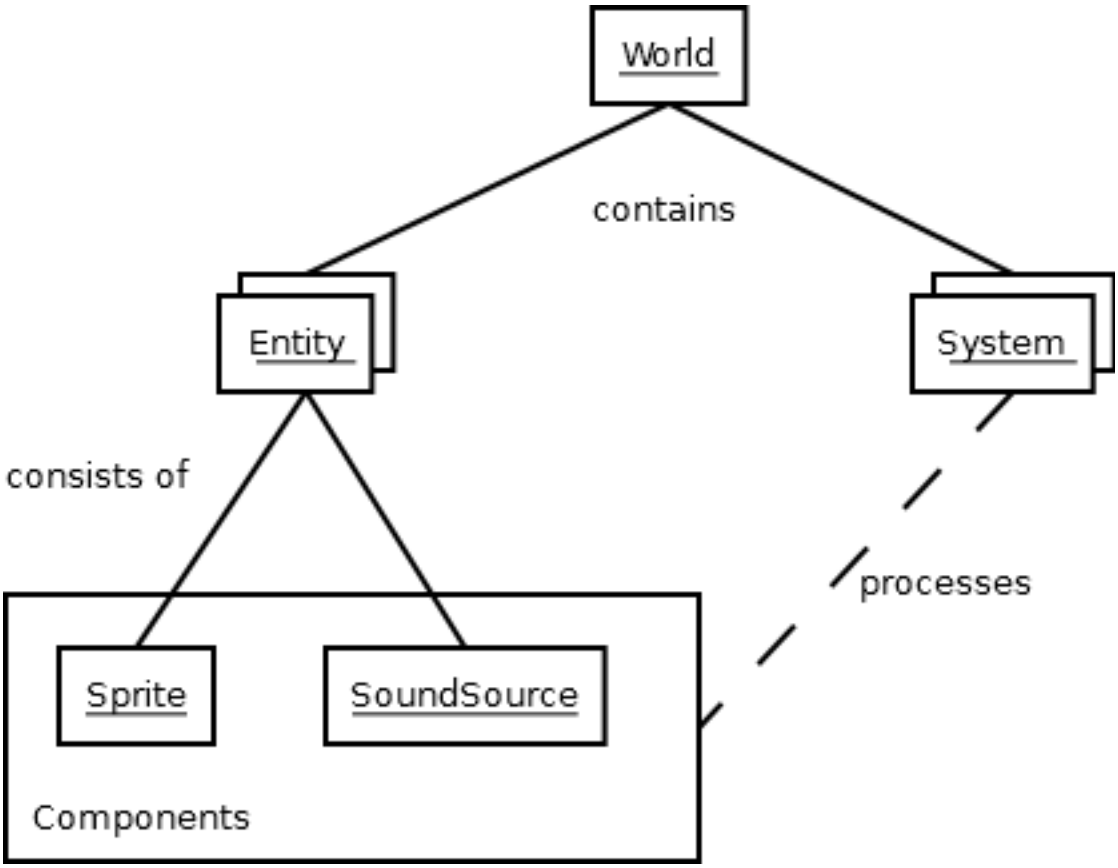
Subclassing with traditional OOP for behavioural changes also might bloat your classes with unnecessary information, causing the memory footprint for your application to rise without any need. Let's assume you have a `Truck` class that inherits from `Car`. Let's further assume that all trucks in your application look the same. Why should any of those carry a `sprite` or `color` attribute? You would need to refactor your `Car` class to get rid of those superfluous information, adding another level of subclassing. If at a later point of time you decide to give your trucks different colors, you need to refactor everything again.

Wouldn't it be easier to deal with colors, if they are available on the truck and leave them out, if they are not? We initially stated that the component-based approach aims to separate data (information) from code logic. That said, if the truck has a color, we can handle it easily, if it has not, we will do as usual.

Also, checking for the color of an object (regardless, if it is a truck, car, airplane or death star) allows us to apply the same or similar behaviour for every object. If the information is available, we will process it, if it is not, we will not do anything.

All in all

Once we split up the previously OOP-style classes into pure data containers and some separate processing code for the behaviour, we are talking about components and (processing) systems. A component is a data container, ideally grouping related information on a granular level, so that it is easy to (re)use. When you combine different components to build your in-application objects and instantiate those, we are talking about entities.



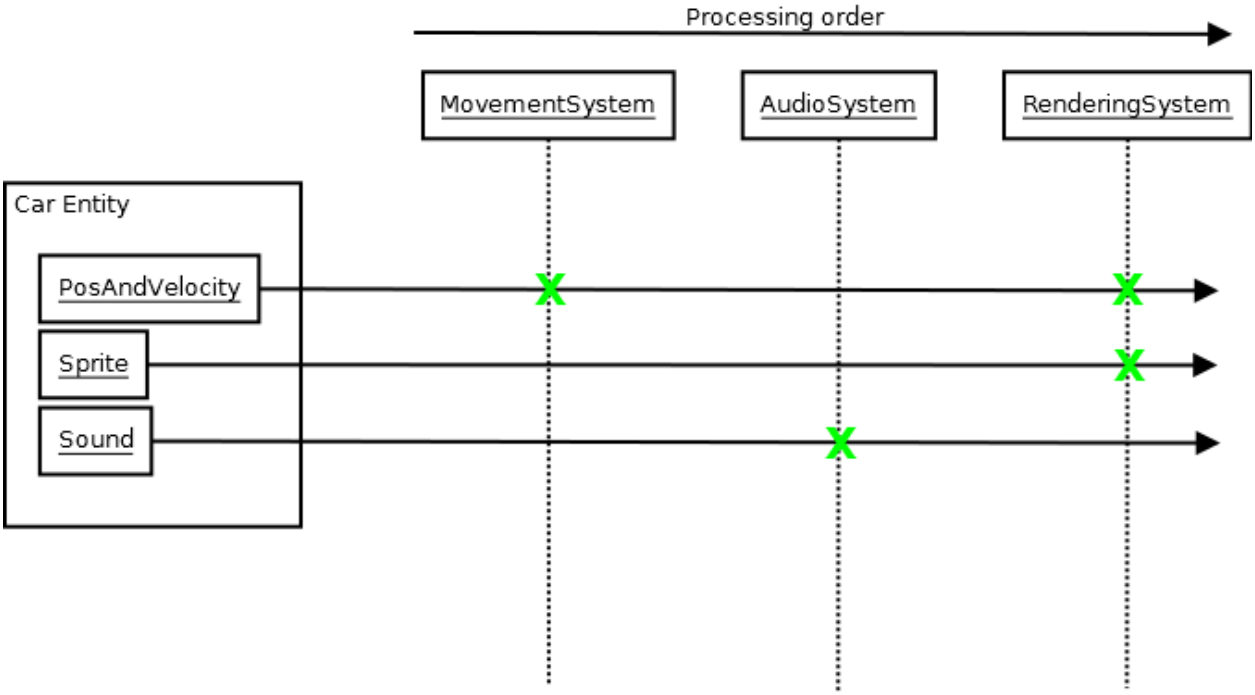
Component provides information (data bag)

Entity In-application instance that consists of *component* items

System Application logic for working with *Entity* items and their *component* data

World The environment that contains the different *System* instances and all *Entity* items with their *component* data

Within a strict COP design, the application logic (ideally) only knows about data to process. It does not know anything about entities or complex classes and only operates on the data.



To keep things simple, modular and easy to maintain and change, you usually create small processing systems, which perform the necessary operations on the data they shall handle. That said, a `MovementSystem` for our car entity would only operate on the position and velocity component of the car entity. It does not know anything about the the car's sprite or sounds that the car makes, since *this is nothing it has to deal with*.

To display the car on the screen, a `RenderSystem` might pick up the sprite component of the car, maybe along with the position information (so it know, where to place the sprite) and render it on the screen.

If you want the car to play sounds, you would add an audio playback system, that can perform the task. Afterwards you can add the necessary audio information via a sound component to the car and it will make noise.

Component-based design with ebs

Note: This section will deal with the specialities of COP patterns and provides the bare minimum of information.

ebs provides a *World* class in which all other objects will reside. The *World* will maintain both, *Entity* and component items, and allows you to set up the processing logic via the *System* and *Applicator* classes.

```
>>> appworld = World()
```

Components can be created from any class that inherits from the `object` type and represent the data bag of information for the entity. and application world. Ideally, they should avoid any application logic (except from getter and setter properties).

```
class Position2D(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

Entity objects define the in-application objects and only consist of component-based attributes. They also require a *World* at object instantiation time.

```
class CarEntity(Entity):
    def __init__(self, world, x=0, y=0):
        self.position2d = Position2D(x, y)
```

Note: The *world* argument in `__init__()` is necessary. It will be passed to the internal `__new__()` constructor of the *Entity* and stores a reference to the *World* and also allows the *Entity* to store its information in the *World*.

The *Entity* also requries its attributes to be named exactly as their component class name, but in lowercase letters. If you name a component `MyAbsolutelyAwesomeDataContainer`, an *Entity* will force you to write the following:

```
class SomeEntity(Entity):
    def __init__(self, world):
        self.myabsolutelyawesomedatacontainer = MyAbsolutelyAwesomeDataContainer()
```

Note: This is not entirely true. A reference of the object will be stored on a per-class-in-mro basis. This means that if `MyAbsolutelyAwesomeDataContainer` inherits from `ShortName`, you can also do:

```
class SomeEntity(Entity):
    def __init__(self, world):
        self.shortname = MyAbsolutelyAwesomeDataContainer()
```

Components should be as atomic as possible and avoid complex inheritance. Since each value of an *Entity* is stored per class in its mro list, components inheriting from the same class(es) will overwrite each other on conflicting classes:

```
class Vector(Position2D):
    def __init__(self, x=0, y=0, z=0):
        super(Vector, self).__init__(x, y)


class SomeEntity(Entity):
    def __init__(self, world):
        # This will associate self.position2d with the new Position2D
        # value, while the previous Vector association is overwritten
        self.position2d = Position2D(4, 4)

        # self.vector will also associate a self.position2d attribute
        # with the Entity, since Vector inherits from Position2D. The
        # original association will vanish, and each call to
        # entity.position2d will effectively manipulate the vector!
        self.vector = Vector(1,2,3)
```

API

class ebs.**Entity**(*world : World*)

An entity is a specific object living in the application world. It does not carry any data or application logic, but merely acts as identifier label for data that is maintained in the application world itself.

As such, it is an composition of components, which would not exist without the entity identifier. The entity itself is non-existent to the application world as long as it does not carry any data that can be processed by a system within the application world.

id

The id of the Entity. Every Entity has a unique id, that is represented by a `uuid.UUID` instance.

world

The *World* the entity resides in.

delete() → None

Deletes the *Entity* from its *World*. This basically calls *World.delete()* with the *Entity*.

class ebs.**Applicator**

A processing system for combined data sets. The *Applicator* is an enhanced *System* that receives combined data sets based on its set *System.componenttypes*

is_applicator

A boolean flag indicating that this class operates on combined data sets.

componenttypes

A tuple of class identifiers that shall be processed by the *Applicator*.

process(*world : World*, *componentsets : iterable*)

Processes tuples of component items. `componentsets` will contain object tuples, that match the *componenttypes* of the *Applicator*. If, for example, the *Applicator* is defined as

```
class MyApplicator(Applicator):
    def __init__(self):
        self.componenttypes = (Foo, Bar)
```

its process method will receive `(Foo, Bar)` tuples

```
def process(self, world, componentsets):
    for foo_item, bar_item in componentsets:
        ...
```

Additionally, the *Applicator* will not process all possible combinations of valid components, but only those, which are associated with the same *Entity*. That said, an *Entity* *must* contain a `Foo` as well as a `Bar` component in order to have them both processed by the *Applicator* (while a *System* with the same `componenttypes` would pick either of them, depending on their availability).

class ebs.**System**

A processing system within an application world consumes the components of all entities, for which it was set up. At time of processing, the system does not know about any other component type that might be bound to any entity.

Also, the processing system does not know about any specific entity, but only is aware of the data carried by all entities.

componenttypes

A tuple of class identifiers that shall be processed by the *System*

process(*world : World*, *components : iterable*)

Processes component items.

This method has to be implemented by inheriting classes.

class ebs.**World**

An application world defines the combination of application data and processing logic and how the data will be processed. As such, it is a container object in which the application is defined.

The application world maintains a set of entities and their related components as well as a set of systems that process the data of the entities. Each processing system within the application world only operates on a certain set of components, but not all components of an entity at once.

The order in which data is processed depends on the order of the added systems.

systems

The processing system objects bound to the world.

add_system(*system : object*)

Adds a processing system to the world. The system will be added as last item in the processing order.

The passed system does not have to inherit from *System*, but must feature a `componenttypes` attribute and a `process()` method, which match the signatures of the *System* class

```
class MySystem(object):
    def __init__(self):
        # componenttypes can be any iterable as long as it
        # contains the classes the system should take care of
        self.componenttypes = [AClass, AnotherClass, ...]

    def process(self, world, components):
        ...
```

If the system shall operate on combined component sets as specified by the *Applicator*, the class instance must contain a `is_applicator` property, that evaluates to `True`

```
class MyApplicator(object):
    def __init__(self):
        self.is_applicator = True
        self.componenttypes = [...]

    def process(self, world, components):
        pass
```

The behaviour can be changed at run-time. The `is_applicator` attribute is evaluated for every call to *World.process()*.

delete(*entity : Entity*)

Removes an *Entity* from the World, including all its component data.

delete_entities(*entities : iterable*)

Removes a set of *Entity* instances from the World, including all their component data.

insert_system(*index : int*, *system : System*)

Adds a processing *System* to the world. The system will be added at the specified position in the processing order.

get_entities(*component : object*) → [Entity, ...]

Gets the entities using the passed component.

Note: This will not perform an identity check on the component but rely on its `__eq__` implementation instead.

---

process()
Processes all component items within their corresponding *System* instances.

remove_system(*system : System*)
Removes a processing *System* from the world.

events - General purpose event handling routines

class events.**EventHandler**(*sender*)
A simple event handling class, which manages callbacks to be executed.

The EventHandler does not need to be kept as separate instance, but is mainly intended to be used as attribute in event-aware class objects.

```
>>> def myfunc(sender):
...     print("event triggered by %s" % sender)
...
>>> class MyClass(object):
...     def __init__(self):
...         self.anevent = EventHandler(self)
...
>>> myobj = MyClass()
>>> myobj.anevent += myfunc
>>> myobj.anevent()
event triggered by <__main__.MyClass object at 0x801864e50>
```

callbacks
A list of callbacks currently bound to the *EventHandler*.

sender
The responsible object that executes the *EventHandler*.

add(*callback : Callable*)
Adds a callback to the *EventHandler*.

remove(*callback : Callable*)
Removes a callback from the *EventHandler*.

__call__(**args*) → [...]
Executes all connected callbacks in the order of addition, passing the *sender* of the *EventHandler* as first argument and the optional args as second, third, ... argument to them.

This will return a list containing the return values of the callbacks in the order of their execution.

class events.**MPEventHandler**(*sender*)
An asynchronous event handling class based on *EventHandler*, in which callbacks are executed in parallel. It is the responsibility of the caller code to ensure that every object used maintains a consistent state. The *MPEventHandler* class will not apply any locks, synchronous state changes or anything else to the arguments or callbacks being used. Consider it a "fire-and-forget" event handling strategy.

Note: The *MPEventHandler* relies on the multiprocessing module. If the module is not available in the target environment, a sdl2.ext.compat.UnsupportedError is raised.

Also, please be aware of the restrictions that apply to the multiprocessing module; arguments and callback functions for example have to be pickable, etc.

__call__(**args*) → AsyncResult
Executes all connected callbacks within a `multiprocessing.pool.Pool`, passing the `sender` as first argument and the optional *args* as second, third, ... argument to them.

This will return a `multiprocessing.pool.AsyncResult` containing the return values of the callbacks in the order of their execution.

resources - Resource management

Every application usually ships with various resources, such as image and data files, configuration files and so on. Accessing those files in the folder hierarchy or in a bundled format for various platforms can become a comple task, for which the *resources* module can provide ideal supportive application components.

The *Resources* class allows you to manage different application data in a certain directory, providing a dictionary-style access functionality for your in-application resources.

Let's assume, your application has the following installation layout

```
Application Directory
    Application.exe
    Application.conf
    data/
        background.jpg
        button1.jpg
        button2.jpg
        info.dat
```

Within the `Application.exe` code, you can - completely system-agnostic - define a new resource that keeps track of all `data` items.

```
apppath = os.path.dirname(os.path.abspath(__file__))
appresources = Resources(os.path.join(apppath, "data"))
# Access some images
bgimage = appresources.get("background.jpg")
btn1image = appresources.get("button1.jpg")
...
```

To access individual files, you do not need to concat paths the whole time and regardless of the current directory, your application operates on, you can access your resource files at any time through the *Resources* instance, you created initially.

The *Resources* class is also able to scan an index archived files, compressed via ZIP or TAR (gzip or bzip2 compression), and subdiectories automatically.

```
Application Directory
    Application.exe
    Application.conf
    data/
        audio/
            example.wav
        background.jpg
        button1.jpg
        button2.jpg
        graphics.zip
            [tileset1.bmp
             tileset2.bmp
             tileset3.bmp
```

```
            ]
        info.dat

tilesimage = appresources.get("tileset1.bmp")
audiofile = appresources.get("example.wav")
```

If you request an indexed file via *Resources.get()*, you will receive a io.BytesIO stream, containing the file data, for further processing.

Note: The scanned files act as keys within the *Resources* class. This means that two files, that have the same name, but are located in different directories, will not be indexed. Only one of them will be accessible through the *Resources* class.

API

class resources.**Resources**([*path=None*[, *subdir=None*[, *excludepattern=None*]]])

The Resources class manages a set of file resources and eases accessing them by using relative paths, scanning archives automatically and so on.

add(*filename : string*)

Adds a file to the resource container. Depending on the file type (determined by the file suffix or name) the file will be automatically scanned (if it is an archive) or checked for availability (if it is a stream or network resource).

add_archive(*filename : string*[, *typehint="zip"*])

Adds an archive file to the resource container. This will scan the passed archive and add its contents to the list of available and accessible resources.

add_file(*filename : string*)

Adds a file to the resource container. This will only add the passed file and do not scan an archive or check the file for availability.

get(*filename : string*) → BytesIO

Gets a specific file from the resource container.

Raises a KeyError, if the *filename* could not be found.

get_filelike(*filename : string*) → file object

Similar to *get()*, but tries to return the original file handle, if possible. If the found file is only available within an archive, a io.BytesIO instance will be returned.

Raises a KeyError, if the *filename* could not be found.

get_path(*filename : string*) → string

Gets the path of the passed *filename*. If *filename* is only available within an archive, a string in the form filename@archivename will be returned.

Raises a KeyError, if the *filename* could not be found.

scan(*path : string[, subdir=None[, excludepattern=None]*)

Scans a path and adds all found files to the resource container. If a file within the path is a supported archive (ZIP or TAR), its contents will be indexed aut added automatically.

The method will consider the directory part (os.path.dirname) of the provided *path* as path to scan, if the path is not a directory. If *subdir* is provided, it will be appended to the path and used as starting point for adding files to the resource container.

excludepattern can be a regular expression to skip directories, which match the pattern.

resources.**open_tarfile**(*archive : string*, *filename : string*[, *directory=None*[, *ftype=None*]]) → BytesIO

Opens and reads a certain file from a TAR archive. The result is returned as `BytesIO` stream. *filename* can be a relative or absolute path within the TAR archive. The optional *directory* argument can be used to supply a relative directory path, under which *filename* will be searched.

ftype is used to supply additional compression information, in case the system cannot determine the compression type itself, and can be either **"gz"** for gzip compression or **"bz2"** for bzip2 compression.

If the filename could not be found or an error occured on reading it, `None` will be returned.

Raises a `TypeError`, if *archive* is not a valid TAR archive or if *ftype* is not a valid value of ("gz", "bz2").

Note: If *ftype* is supplied, the compression mode will be enforced for opening and reading.

---

resources.**open_url**(*filename : string*[, *basepath=None*]) → file object

Opens and reads a certain file from a web or remote location. This function utilizes the `urllib2` module for Python 2.7 and `urllib` for Python 3.x, which means that it is restricted to the types of remote locations supported by the module.

basepath can be used to supply an additional location prefix.

resources.**open_zipfile**(*archive : string*, *filename : string*[, *directory : string*]) → BytesIO

Opens and reads a certain file from a ZIP archive. The result is returned as `BytesIO` stream. *filename* can be a relative or absolute path within the ZIP archive. The optional *directory* argument can be used to supply a relative directory path, under which *filename* will be searched.

If the filename could not be found, a `KeyError` will be raised. Raises a `TypeError`, if *archive* is not a valid ZIP archive.

scene - Scene management

class scene.**SceneManager**

The SceneManager takes care of scene transitions, preserving scene states and everything else to maintain and ensure the control flow between different scenes.

name

The name of the *Scene*.

scenes

The scene stack.

next

The next *Scene* to run on calling *update()*.

current

The currently running/active *Scene*.

switched

A `sdl2.ext.EventHandler` that is invoked, when a new *Scene* is started.

push(*scene : Scene*) → None

Pushes a new *Scene* to the scene stack.

The *current* scene will be put on the scene stack for later execution, while the passed *scene* will be set as current one. Once the newly pushed scene has ended or was paused, the previous scene will continue its execution.

pop() → None
Pops a scene from the scene stack, bringing it into place for being executed on the next update.

pause() → None
Pauses the *current* scene.

unpause() → None
Continues the *current* scene.

update() → None
Updates the scene state and switches to the next scene, if any has been pushed into place.

class scene.**Scene**([*name=None*])
A simple scene state object used to maintain the application workflow based on the presentation of an application.

manager
The *SceneManager*, the *Scene* is currently executed on.

Note: This will be set automatically on starting the *Scene* by the *SceneManager*. If the *Scene* is ended, it will be reset.

---

state
The current scene state.

started
A sdl2.ext.EventHandler that is invoked, when the *Scene* starts.

paused
A sdl2.ext.EventHandler that is invoked, when the *Scene* is paused.

unpaused
A sdl2.ext.EventHandler that is invoked, when the *Scene* is unpaused.

ended
A sdl2.ext.EventHandler that is invoked, when the *Scene* ends.

is_running
Indicates, if the scene is currently running.

is_paused
Indicates, if the scene is currently paused.

has_ended
Indicates, if the scene has ended.

start() → None
Starts the *Scene*. If the *Scene* is running or paused, nothing will be done.

pause() → None
Pauses the *Scene*. If the *Scene* is not running, nothing will be done.

unpause() → None
Continues the *Scene*. If the *Scene* is not paused, nothing will be done.

end() → None
Ends the *Scene*. If the *Scene* has ended already, nothing will be done.

sysfont - Font detection helpers

The *sysfont* module enables you to find fonts installed on the underlying operating system. It supports Win32 and fontconfig-based (most Unix-like ones, such as Linux or BSD) systems.

sysfont.**STYLE_NORMAL**
Indicates a normal font style.

sysfont.**STYLE_BOLD**
Indicates a bold font style.

sysfont.**STYLE_ITALIC**
Indicates an italic font style.

sysfont.**init**() → None
Initializes the internal font cache. This does not need to be called explicitly. It is called automatically, if one of the retrieval functions is executed for the first time.

sysfont.**get_font**(*name : string[, style=STYLE_NORMAL[, ftype=None]]) -> (str, str, int, str, str*)
Retrieves the best matching font file for the given *name* and criteria. The return value will be a, containing the following information: *(family, font name, font style, font type, filename)*

- family: string, denotes the font family
- font name: string, the name of the font
- font style: int, a combination of the different STYLE_ values
- font type: string, the font file type (e.g. TTF, OTF, ...)
- filename: the name of the physical file

If no font could be found, None will be returned.

sysfont.**get_fonts**(*name : string[, style=STYLE_NORMAL[, ftype=None]]) -> ((str, str, int, str, str), ...*)
Retrieves all fonts matching the given family or font name, *style* and, if provided, font file type. The return values will be tuples, containing the following information: *(family, font name, font style, font type, filename)*

- family: string, denotes the font family
- font name: string, the name of the font
- font style: int, a combination of the different STYLE_ values
- font type: string, the font file type (e.g. TTF, OTF, ...)
- filename: the name of the physical file

If no font could be found, None will be returned.

sysfont.**list_fonts**() → iterator
Retrieves an iterator over all found fonts. The values of the iterator will be tuples, containing the following information: *(family, font name, font style, font type, filename)*

- family: string, denotes the font family
- font name: string, the name of the font
- font style: int, a combination of the different STYLE_ values
- font type: string, the font file type (e.g. TTF, OTF, ...)
- filename: the name of the physical file

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Documentation TODOs

Last generated on: Sep 18, 2017

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I