Contents

[Asking for Help 2](#_Toc8661552)

[PYTHON @ HIGHER LEVEL 2](#_Toc8661553)

[REPEATABLE ENVIRONMENT 2](#_Toc8661554)

[CI/CD 2](#_Toc8661555)

[Testing 2](#_Toc8661556)

[STRATEGY for Data Science 2](#_Toc8661557)

[More ON Testing 2](#_Toc8661558)

[PYTEST Fixtures 2](#_Toc8661559)

[PYTEST PLUGINS Coverage 2](#_Toc8661560)

[Test Driven Development 3](#_Toc8661561)

[MOCK 3](#_Toc8661562)

[MOTO 3](#_Toc8661563)

[TEST STRUCTURE 3](#_Toc8661564)

[TEST MATRIX 3](#_Toc8661565)

[CONFIG 3](#_Toc8661566)

[PIPENV 3](#_Toc8661567)

[Environments, Apps, Libraries 3](#_Toc8661568)

[Virtual Environments 3](#_Toc8661569)

[What are your REQS? 4](#_Toc8661570)

[Why PIPENV? 4](#_Toc8661571)

[CONDA 4](#_Toc8661572)

[Libraries versus Applications 4](#_Toc8661573)

[SEMVER 4](#_Toc8661574)

[REQUIREMENTS w.r.t SEMVER 4](#_Toc8661575)

[Tracking the version 4](#_Toc8661576)

[Keep Version Consistent 4](#_Toc8661577)

[SETUPTOOLS\_SCM 4](#_Toc8661578)

[DOCKER 4](#_Toc8661579)

[Build vs. Mounting 4](#_Toc8661580)

[Build vs. Runtime 4](#_Toc8661581)

[LOGGING 5](#_Toc8661582)

[Functional Programming 5](#_Toc8661583)

[STATE 5](#_Toc8661584)

[DECLARATIVE vs. PROCEDURAL 5](#_Toc8661585)

[HASH 5](#_Toc8661586)

[SALTED GRAPHS 5](#_Toc8661587)

[Data Dependency Hell 5](#_Toc8661588)

[Content Addressable Filesystems 5](#_Toc8661589)

[Versioned DAG 5](#_Toc8661590)

[Minimal Implementation 5](#_Toc8661591)

[… With Considerations 5](#_Toc8661592)

[The Signature Input 6](#_Toc8661593)

[Custom Salts 6](#_Toc8661594)

[Data-Driven Salts 6](#_Toc8661595)

[LUIGI 6](#_Toc8661596)

[Atomicity 6](#_Toc8661597)

[Tasks 6](#_Toc8661598)

[Task Skeleton 6](#_Toc8661599)

[Parameters 7](#_Toc8661600)

[Atomic Targets 7](#_Toc8661601)

[Task Types 7](#_Toc8661602)

[External Programs 7](#_Toc8661603)

[Output 7](#_Toc8661604)

[Requirements 7](#_Toc8661605)

[APACHE PARQUET/ARROW 7](#_Toc8661606)

[Column Stores 8](#_Toc8661607)

[Key-Value Stores 8](#_Toc8661608)

[Column Stats 8](#_Toc8661609)

[FASTPARQUET V. PYARROW 8](#_Toc8661610)

[DASK 8](#_Toc8661611)

[Result in Pieces 8](#_Toc8661612)

[Split, Apply, Combine 8](#_Toc8661613)

[Dask.DataFrame 8](#_Toc8661614)

[Predicates 8](#_Toc8661615)

[Indexed DataFrames – Surrogate keys 8](#_Toc8661616)

[Dask Debugging 9](#_Toc8661617)

[Synchronous Debugging 9](#_Toc8661618)

[DJANGO 9](#_Toc8661619)

[2 Types of Data Use 9](#_Toc8661620)

[ORM (Object Relational Mapping) 9](#_Toc8661621)

[Inheritable Models 9](#_Toc8661622)

[Composable Models 9](#_Toc8661623)

[Migrations 9](#_Toc8661624)

[Page Lifecycle 9](#_Toc8661625)

[Django Views 9](#_Toc8661626)

[MetaProgramming 9](#_Toc8661627)

[MetaClasses 10](#_Toc8661628)

[Model Options 10](#_Toc8661629)

[ModelBase 10](#_Toc8661630)

[API 10](#_Toc8661631)

[ORM as a Service 10](#_Toc8661632)

[Django REST Framework 10](#_Toc8661633)

[Serializers 10](#_Toc8661634)

[Views 10](#_Toc8661635)

[Django REST Witchcraft 10](#_Toc8661636)

[Swagger 10](#_Toc8661637)

[API and Clients 10](#_Toc8661638)

[HTTP Methods 10](#_Toc8661639)

[API Views 10](#_Toc8661640)

[Client: Requests 11](#_Toc8661641)

[DRF CoreAPI 11](#_Toc8661642)

[Mutable Objects 11](#_Toc8661643)

[Pass By Ref or Value 11](#_Toc8661644)

[Mutability vs State 11](#_Toc8661645)

[Immutable Workflows 11](#_Toc8661646)

[Deterministic Workflows 11](#_Toc8661647)

[Not Every Workflow is Deterministic 11](#_Toc8661648)

[Snapshots 11](#_Toc8661649)

[Atomicity 12](#_Toc8661650)

[Atomic Writes 12](#_Toc8661651)

[Database 12](#_Toc8661652)

[Luigi Targets 12](#_Toc8661653)

[Normalization 12](#_Toc8661654)

[Stars and Flakes 12](#_Toc8661655)

[Intelligentg Keys 12](#_Toc8661656)

[Atomics 12](#_Toc8661657)

[Memoization 12](#_Toc8661658)

[Memoization with Decorators 12](#_Toc8661659)

[Lazy Properties 13](#_Toc8661660)

[Global Cache 13](#_Toc8661661)

[Instance Cache 13](#_Toc8661662)

[Optimization 13](#_Toc8661663)

[Memory Views 13](#_Toc8661664)

[GIL and Parallelism 14](#_Toc8661665)

[Vizualization 14](#_Toc8661666)

[Grammar Graphics 14](#_Toc8661667)

[Color and Data Science 14](#_Toc8661668)

[Colorspace 14](#_Toc8661669)

[Qualitative Color - ColorBrewer 14](#_Toc8661670)

[Sequential Color 14](#_Toc8661671)

[Perceptually Uniform Maps 14](#_Toc8661672)

[Divergent 15](#_Toc8661673)

[Vizualization as Map Reduce 15](#_Toc8661674)

[DECORATOR 16](#_Toc8661675)

[Function as Decorator 16](#_Toc8661676)

[Function as Decorator (w/o arguments) 16](#_Toc8661677)

[Function as Decorator (w/ arguments) 16](#_Toc8661678)

[Class as Decorator 16](#_Toc8661679)

[Class as Decorator (w/o arguments) 16](#_Toc8661680)

[Class as Decorator (w/ arguments) 16](#_Toc8661681)

[REGISTRIES 16](#_Toc8661682)

[Registering Plugins 16](#_Toc8661683)

[Using @wraps with decorator 16](#_Toc8661684)

[Functional decorator 17](#_Toc8661685)

[Decorator In class style 17](#_Toc8661686)

[CONTEXT MANAGER 17](#_Toc8661687)

[Context Manager as a Class 17](#_Toc8661688)

[Handling Exceptions 17](#_Toc8661689)

[Context Manager as Generator 17](#_Toc8661690)

[General Form 17](#_Toc8661691)

[Sequence 17](#_Toc8661692)

[ITERATOR 17](#_Toc8661693)

[Array 17](#_Toc8661694)

[Generator 17](#_Toc8661695)

[Map 18](#_Toc8661696)

[Reduce 18](#_Toc8661697)

[Zip 18](#_Toc8661698)

[Filter 18](#_Toc8661699)

[CLASS 18](#_Toc8661700)

[Unpacking (\*args and \*\*kwargs) 18](#_Toc8661701)

[Unpacking argument lists 18](#_Toc8661702)

[kwarg (keyword argument) 18](#_Toc8661703)

[args and \*kwargs together 18](#_Toc8661704)

[General 18](#_Toc8661705)

[Inherit and Override 18](#_Toc8661706)

[Abstracts and Interfaces 18](#_Toc8661707)

[Informal 18](#_Toc8661708)

[Formal: use ABC 18](#_Toc8661709)

[Mixins 18](#_Toc8661710)

[Composition 18](#_Toc8661711)

[Composition vs. Inheritance 19](#_Toc8661712)

[Composition as Instance Properties 19](#_Toc8661713)

[Composition as Declarative Classes 19](#_Toc8661714)

[Declarative Classes 19](#_Toc8661715)

[Instance, Class, Static Methods 19](#_Toc8661716)

[Class Method example 19](#_Toc8661717)

[Properties 19](#_Toc8661718)

[@property 19](#_Toc8661719)

[@property is descriptor 19](#_Toc8661720)

[CLASS as Constructors 20](#_Toc8661721)

[Creation Design Patterns 20](#_Toc8661722)

[Simple Factories 20](#_Toc8661723)

[Prototype Factories 20](#_Toc8661724)

[Factory Methods 20](#_Toc8661725)

[Builders 20](#_Toc8661726)

[Abstract Factories 20](#_Toc8661727)

[Factory Boy 20](#_Toc8661728)

[Faker 20](#_Toc8661729)

[Python 21](#_Toc8661730)

[Testing 21](#_Toc8661731)

[Workflows 21](#_Toc8661732)

[Higher Levels 21](#_Toc8661733)

[Deployment 21](#_Toc8661734)

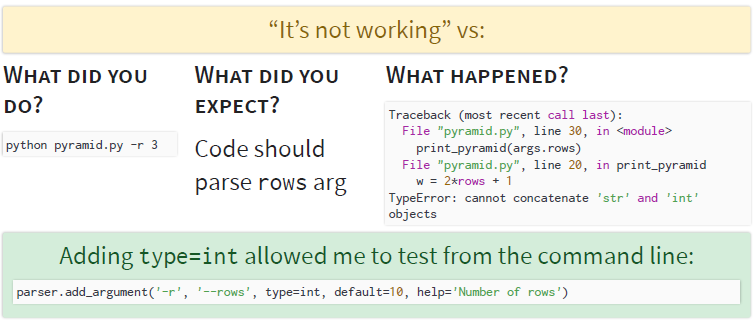
[Looping 21](#_Toc8661735)

[Functional Coding 21](#_Toc8661736)

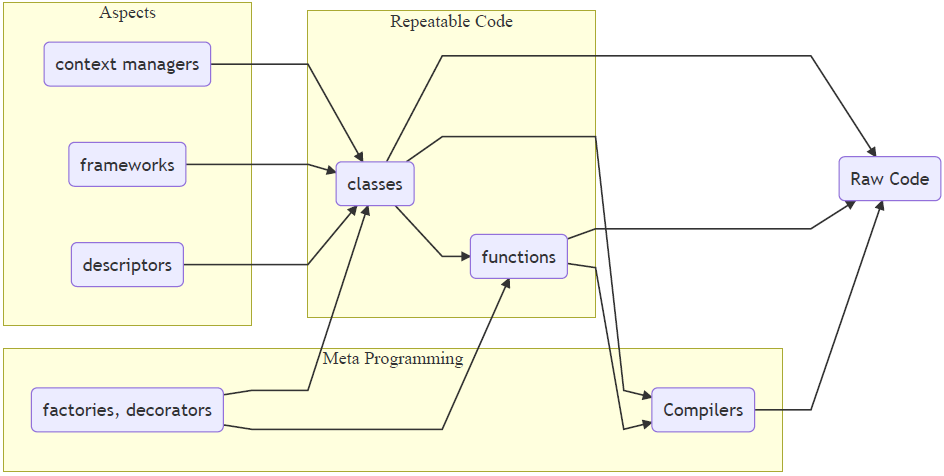
[Composition 21](#_Toc8661737)

[Graphical Programs 21](#_Toc8661738)

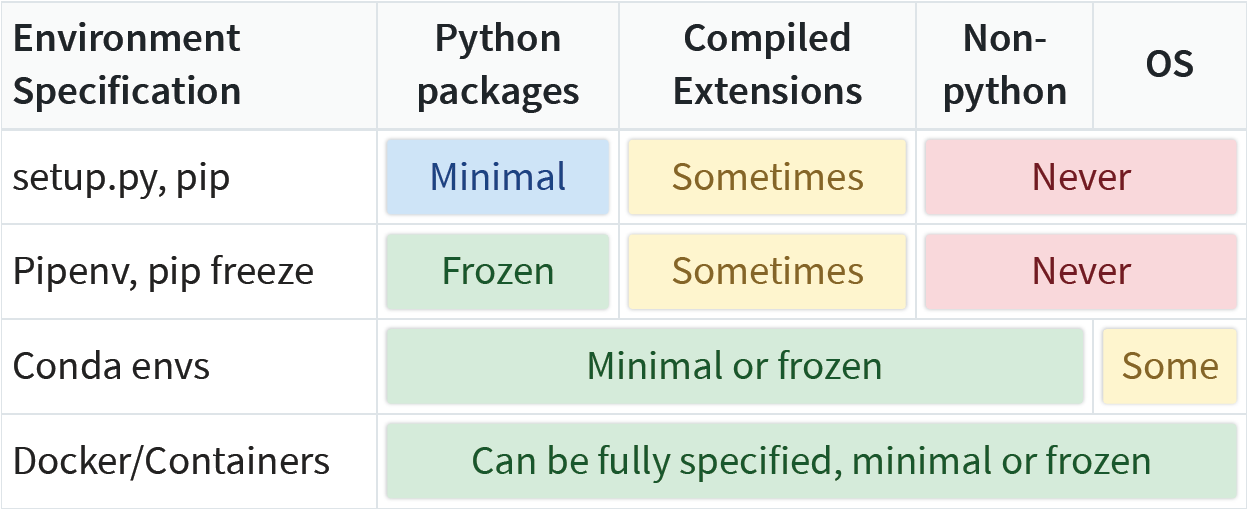
# Asking for Help



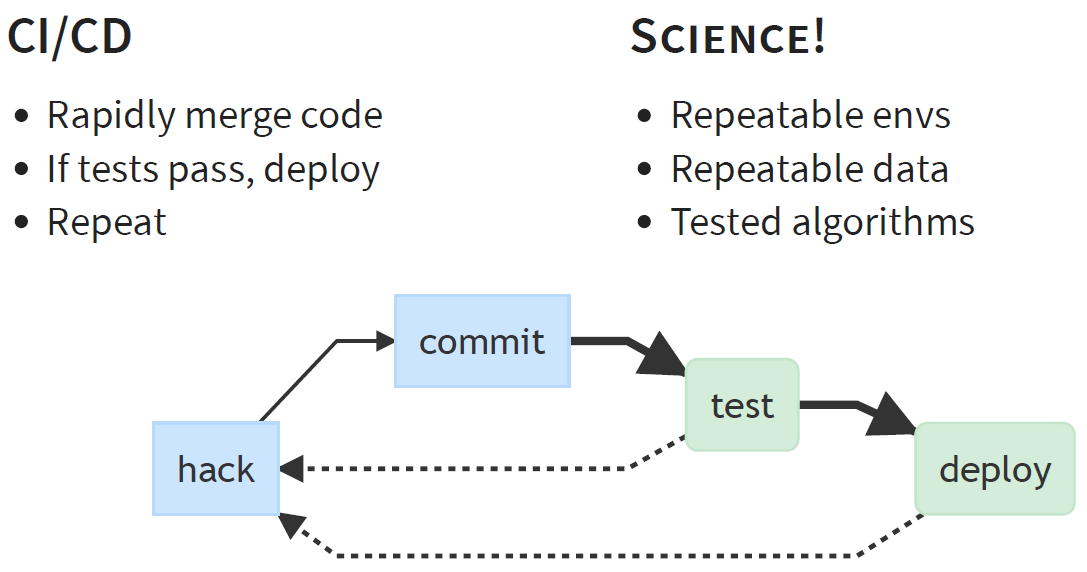
# PYTHON @ HIGHER LEVEL



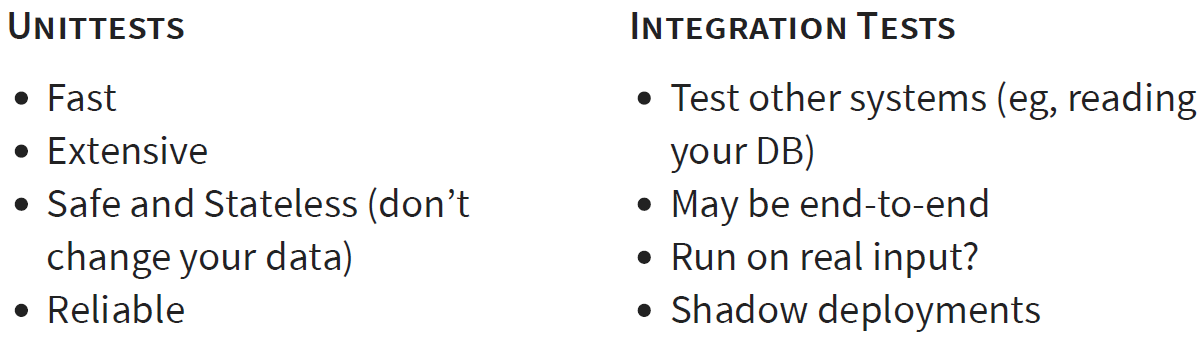
# REPEATABLE ENVIRONMENT



# CI/CD



# Testing

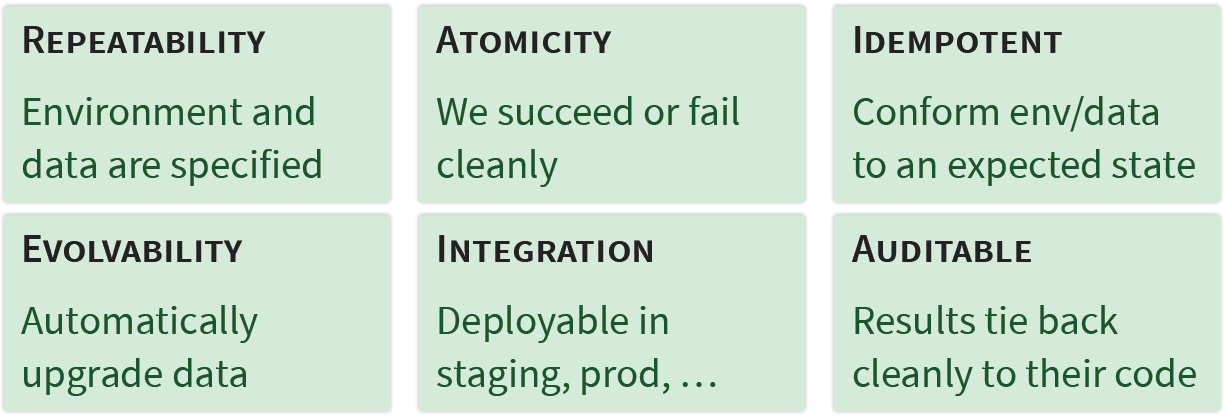


But Science is special

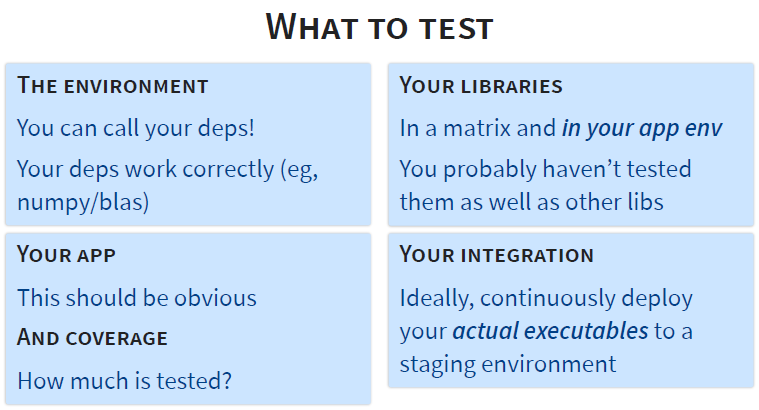
|  |  |
| --- | --- |
| * More complicated environments * Stateful, historical data * Persisted results * Probabilistic algorithms (random failures) | * ‘Qualitative’ results (eg, is model accuracy good enough?) * Evolving input data schemas * Hard to get representative data * Integrated, not isolated - relies on many services |

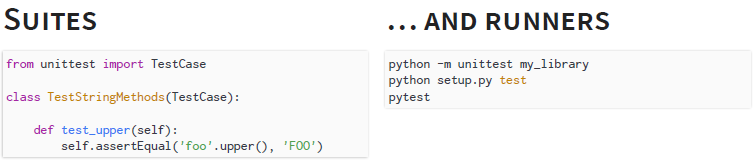
Therefore ….

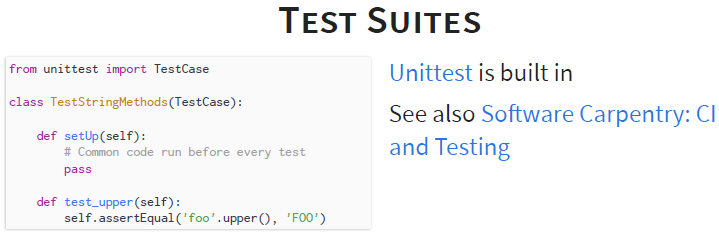
# STRATEGY for Data Science



# More ON Testing







## PYTEST Fixtures

Isolating the setup code for the connection in a fixture is good orthogonal design - don’t need multiple TestCase subclasses in various setUp methods

# content of ./test\_smtpsimple.py

**import** **pytest**

**@pytest**.fixture

**def** **smtp\_connection**():

**import** **smtplib**

**return** smtplib.SMTP("smtp.gmail.com", **587**, timeout=**5**)

**def** **test\_ehlo**(smtp\_connection):

response, msg = smtp\_connection.ehlo()

**assert** response == **250**

**assert** **0** # for demo purposes

… But Fixtures introspect the argument name in a test function and try to find a fixture to run to inject that variable. This violates ‘explicit is better than implicit’ and makes the test cases fragile (can’t refactor the fixture safely). Alternatively,

**from** **functools** **import** wraps

**def** **fixture**(provider, kwarg=**None**):

kwarg = kwarg **or** provider.\_\_name\_\_

**def** **decorator**(func):

**@wraps**(func)

**def** **wrapped**(\*args, \*\*kwargs):

kwargs[kwarg] = provider()

**return** func(\*args, \*\*kwargs)

**return** wrapped

**return** decorator

**def** **smtp\_connection**():

...

**class** **SmtpTests**(TestCase):

**@fixture**(smtp\_connection)

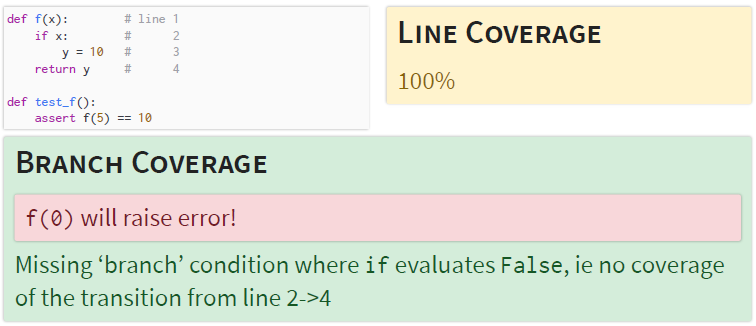
**def** **test\_ehlo**(self, smtp\_connection=**None**):

response, msg = smtp\_connection.ehlo()

...

## PYTEST PLUGINS Coverage

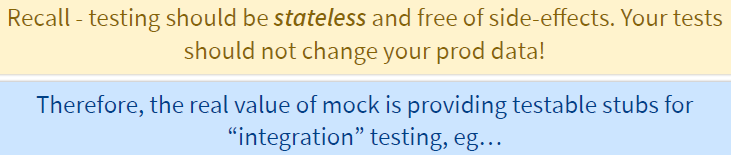
Branch coverage is far more important than line coverage. It looks for tests on every logical branch code could follow.



## Test Driven Development



## MOCK



## MOTO

Moto is Mock for Boto (AWS client). @mock\_s3 allows for basic ‘integration’ testing to one of the most important remote file systems in existence. It doesn’t

work for big data, but it’s killer to make sure your code works!

**import** **boto**

**from** **moto** **import** mock\_s3

**from** **mymodule** **import** MyModel

**@mock\_s3**

**def** **test\_my\_model\_save**():

conn = boto.connect\_s3()

# Need to create bucket since this is all in Moto's 'virtual' AWS account

conn.create\_bucket('harvard')

model\_instance = MyModel('scott', 'is awesome')

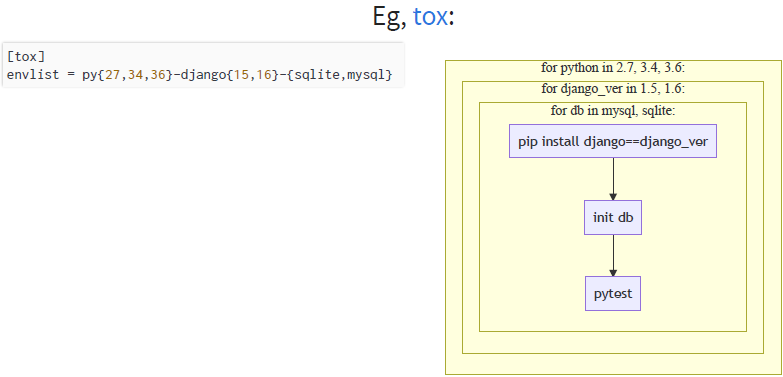
model\_instance.save()

**assert** conn.get\_bucket ('harvard').get\_key('scott').get\_contents\_as\_string() == 'is cool'

## TEST STRUCTURE



## TEST MATRIX



# CONFIG

**The Twelve-Factor App**: *A litmus test for whether an app has all config correctly factored out of the code is whether the codebase could be made open source at any moment, without compromising any credentials*.

You can use relative locations for code deploys - eg within the repo ./data, which is git ignored. This isn’t perfect - it doesn’t work for libraries - but on deploy you can symlink these to other locations if necessary. Read a string value:

**import** **os**

secret = os.environ.get('MY\_APP\_SECRET') **or** ''

or parse it if you want an int, bool, etc

# Choose either python or json syntax

**from** **ast** **import** literal\_eval # Never 'eval'

MY\_FLAG = bool( # Handle 0, False, 1, True, etc

literal\_eval( # Or json.loads

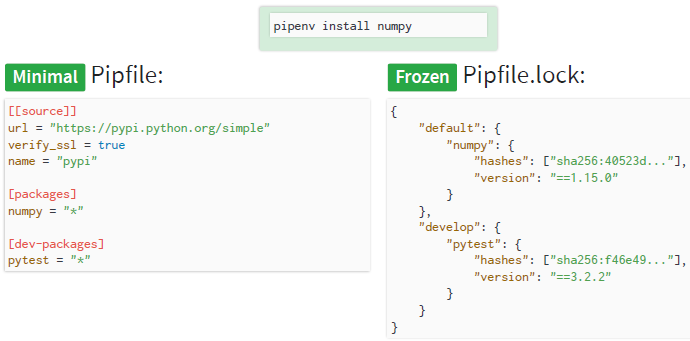
os.environ.get('MY\_APP\_FLAG') **or** '0'

)

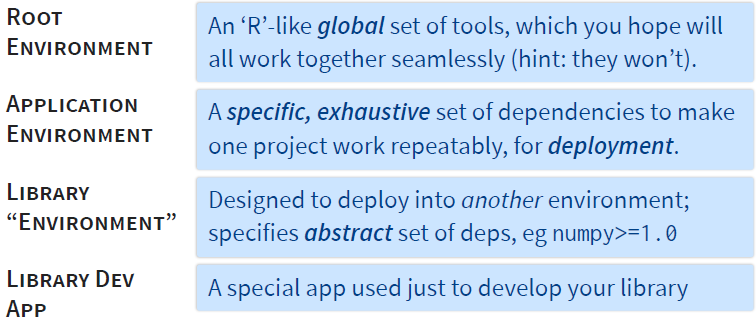
)

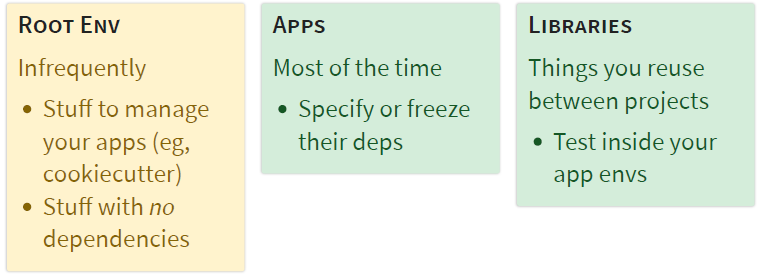
Not all configs secrets. Distinguish between ‘**deploy configs**’ (points to servers, files) and ‘**code configs**’ (runs with verbose debugging, or higher training epochs). Version control code config (or set profiles in code, etc) since they are likely shareable between deploys. Use a **.env** file or **env directory** to specify env variables. **Pipenv** and **docker** automatically read these, and other tools like **envdir** can load them explicitly. These files should not be in VCS! Other tools prefer **INI** or **yaml** systems - eg python’s built in **ConfigParser** and **Luigi**: great composability (eg a system file for servers/IP’s, local file for code config in VCS) and even variables in the configs BUT often not overrideable via the environment.

# PIPENV

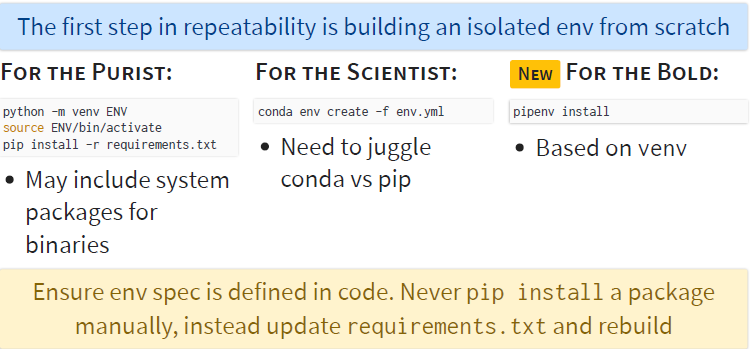


## Environments, Apps, Libraries

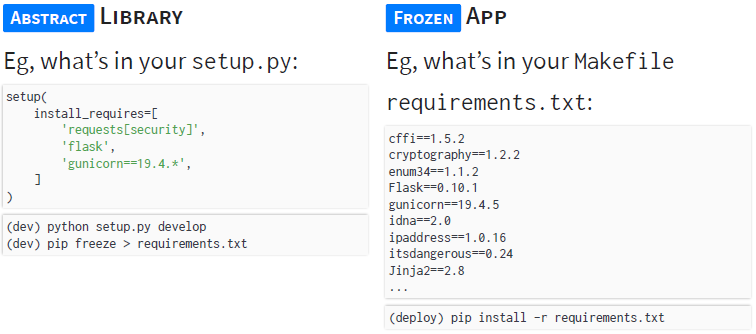




## Virtual Environments



## What are your REQS?



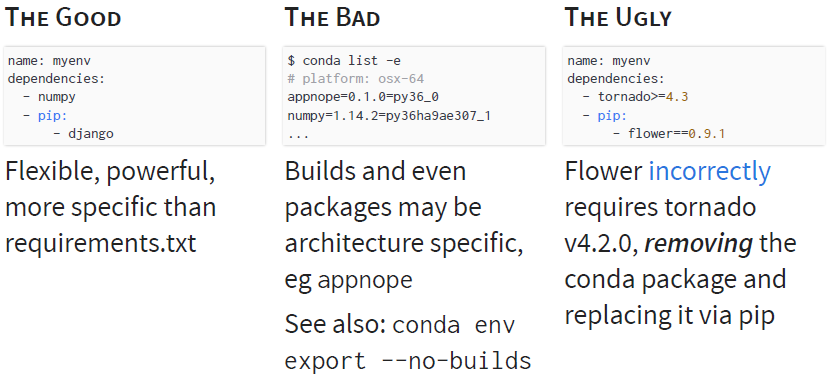
## Why PIPENV?

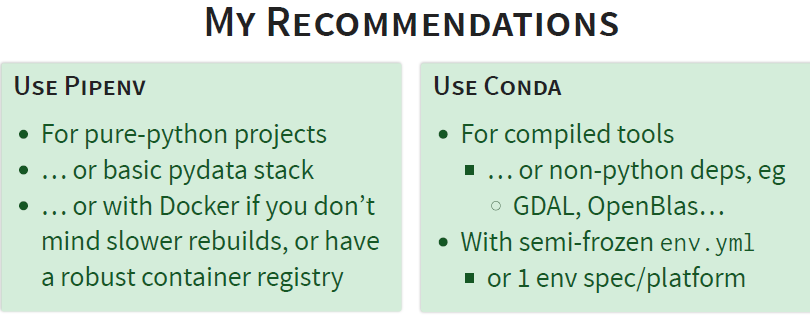


pipenv update

… one-stop upgrade of lock w/o breaking abstract deps

## CONDA

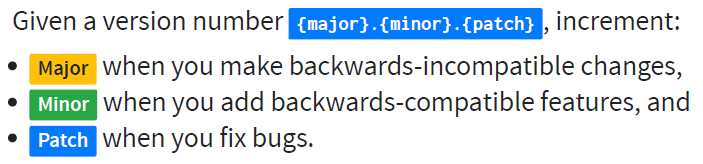




## Libraries versus Applications

* Libraries: define **abstract dependencies** via install\_requires in setup.py. Don’t make decision on version and dependency!
* Applications: define **dependencies and where to get them** in Pipfil*e.* Use this file to update **concrete dependencies** in Pipfile.lock..
* Projects for which library and application isn’t clear. Use install\_requires alongside Pipenv and Pipfile. Tell Pipenv to lock setup.py–declared dependencies with: $pipenv install -e .

# SEMVER



## REQUIREMENTS w.r.t SEMVER

* Should be safe to require x.y.\*
* Specify minor version you need, e.g. >=1.2
* Specify major version, e.g. =3.\*
  + Or, hope for the best. Apps are frozen, and if you don’t know there’s an issue in the next version, it’s okay not to protect against it

## Tracking the version

**THE INSTALL**

# What's set in setup.py

**import** **pkg\_resources**

pkg\_resources.get\_distribution("scipy").version

# site-packages/

# └── scipy-1.1.0-py3.6.egg-info/

# └── PKG-INFO

**THE CODE**

**import** **scipy**

**print**(scipy.\_\_version\_\_) # By convention

# site-packages/

# └── scipy/

# ├── \_\_init\_\_.py

# └── version.py

## Keep Version Consistent

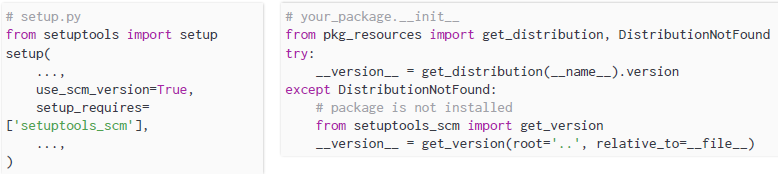
* Need to find a way to specify version once and get it right
* Need version to be globally unique
* Need to account for dev versions for your own code

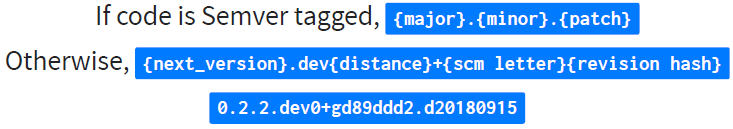
Simple: specify via hardcode in version.py and import in setup.py. But:

* Sometimes import issues
  + Import from your code before setup\_requires packages are installed is risky!
* Prone to forgetting to commit version number changes
* Doesn’t specify commit for dev versions

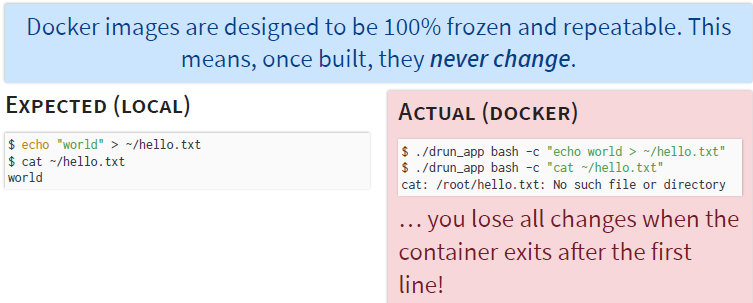
## SETUPTOOLS\_SCM

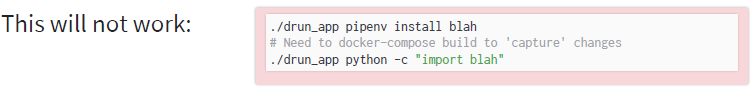
Manage your versions by scm tags



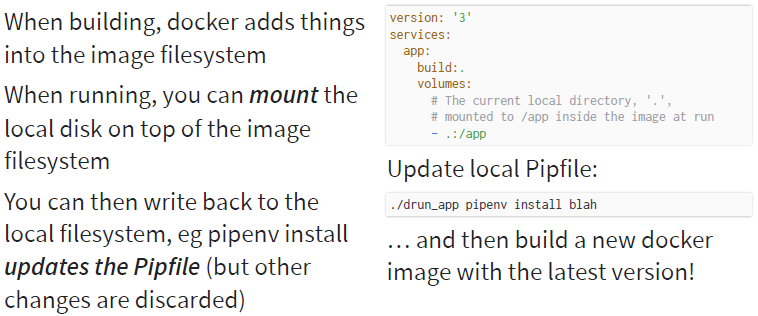


# DOCKER

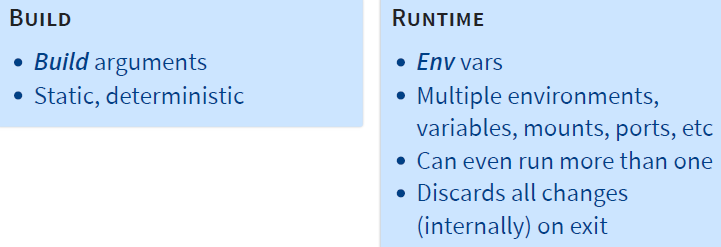


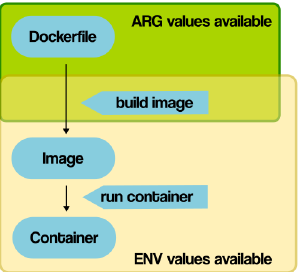


## Build vs. Mounting



## Build vs. Runtime





[Docker ARG, ENV and .env - a Complete Guide](https://vsupalov.com/docker-arg-env-variable-guide/)

# LOGGING

**Use for reqs, auditing, etc**

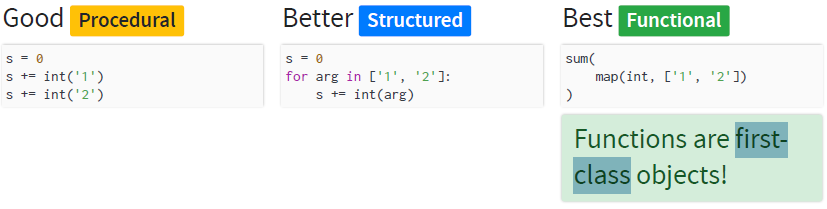
**from** **logging** **import** root

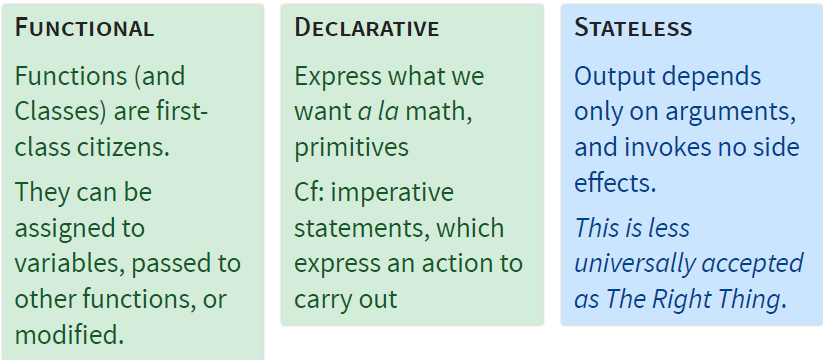
**import** **my\_app**

root.info("Running {} v{}".format(my\_app.\_\_name\_\_, my\_app.\_\_version\_\_))

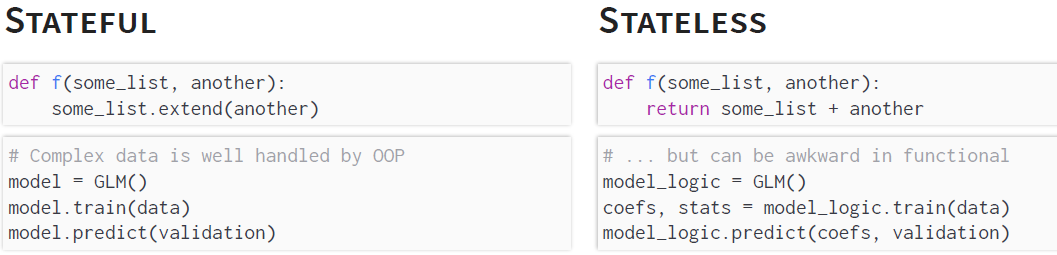
# Functional Programming

Try to write every program as an operation on data, cf:

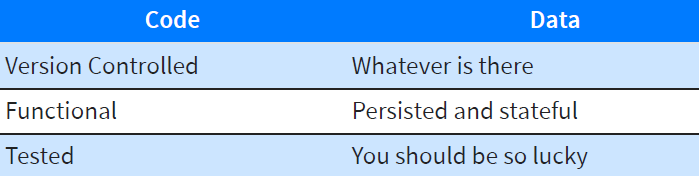




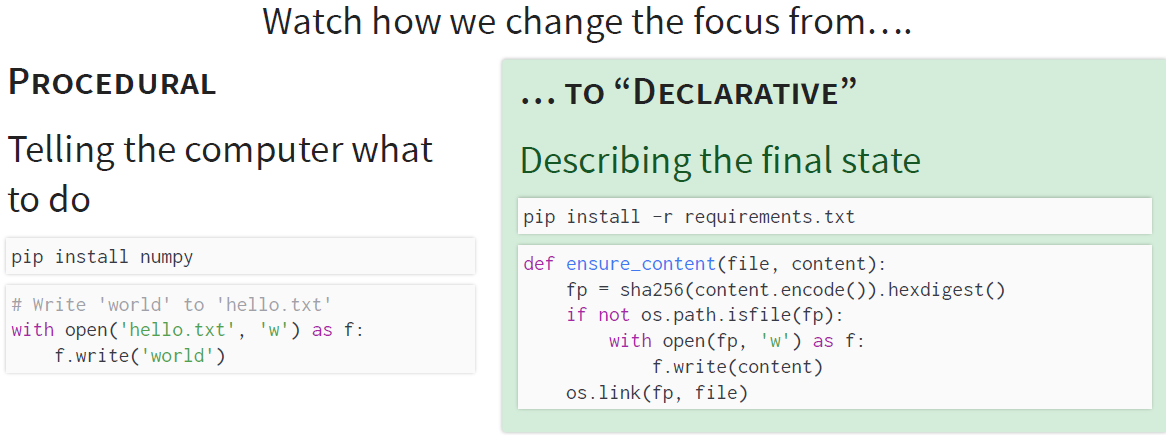
## STATE



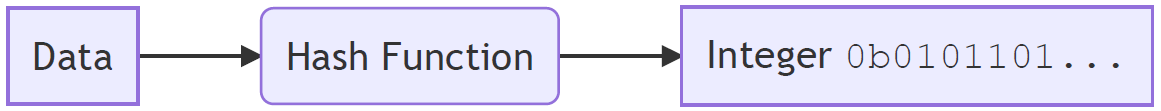
**Sorry State of Stateful Data! Changes in code not reflected in stateful data**



## DECLARATIVE vs. PROCEDURAL



# HASH

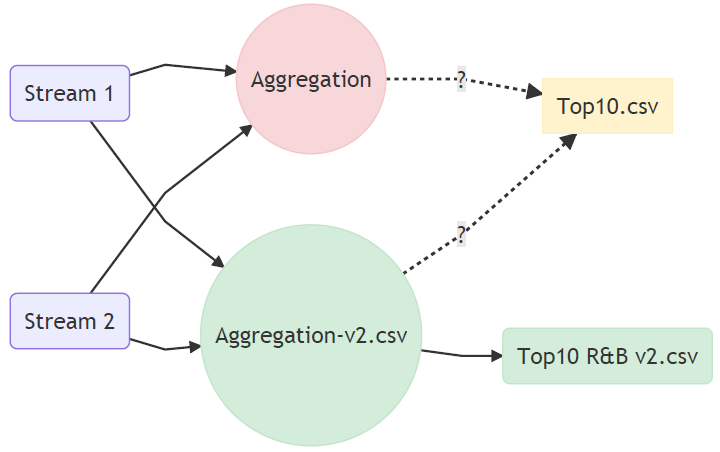


🞟**Hash function**, e.g. sha256, maps arbitrary string of data into fixedwidth integer, e.g. 256 bits 🞟Integer appears uniform and is deterministic 🢥 useful for many algorithms… 🞟Hashes distribute lookups evenly

# SALTED GRAPHS

## Data Dependency Hell

We can fix and rename all downstream file targets to resolve the issue. However, if we forget to rename a downstream target’s output, we have no idea (other than inspecting logs, if we’re lucky) which version of the code and data was used to generate that data. Also, we assume that the authors of every downstream task are aware of the upstream bug and know to handle this in the first place!



## Content Addressable Filesystems

**def** **write**(file\_content):

# Use the hash as the filename!

file\_name = sha256(file\_content.encode()).hexdigest()

**with** atomic\_write(filename) **as** f:

f.write(file\_content)

**GIT commits** are files in a *content addressable file system*. They contain info about the state of the working directory (the ‘tree’), some metadata, plus a ‘pointer’ to the filename - aka, the hash - of the parent commit. By including the hash of the parent commit in it’s own metadata, the hash of a current commit yields a deterministic identifier that uniquely distinguishes it from all other code and all other commit histories, even for the same current code. Therefore, the entire lineage and current state is represented by the commit id.

## Versioned DAG

**def** **get\_salted\_version**(task):

"""Create a salted id/version for this task and lineage"""

msg = ""

# Salt with lineage

**for** req **in** flatten(task.requires()):

msg += get\_salted\_version(req)

# Uniquely specify this task

msg += ','.join([

task.\_\_class\_\_.\_\_name\_\_,

task.\_\_version\_\_,

] + [

f'{pname}={pvalue}'

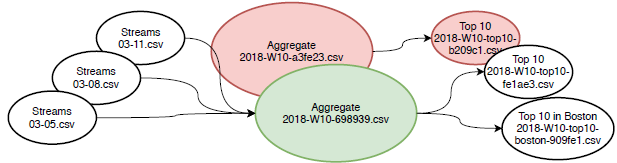
**for** pname, pvalue **in** task.param\_kwargs.items()

]

)

**return** sha256(msg.encode()).hexdigest()

Write the data version into the target path!



### Minimal Implementation

**class** **AggregateArtists**(Task):

\_\_version\_\_ = '1.2 - bugfix'

date\_interval = DateIntervalParameter()

**def** **output**(self):

salt = get\_salted\_version(self)[:**6**]

**return** salted\_target(self, f"aggregate-{salt}.parquet")

...

**def** **get\_salted\_version**(task):

msg = ""

# Salt with lineage

**for** req **in** flatten(task.requires()):

msg += get\_salted\_version(req)

# Uniquely specify this task

msg += ','.join([

task.\_\_class\_\_.\_\_name\_\_,

task.\_\_version\_\_,

] + [

'{}={}'.format(

param\_name,

repr(task.param\_kwargs[param\_name]))

**for** param\_name, param **in** sorted(task.get\_params())

**if** param.significant

]

)

**return** sha256(msg.encode()).hexdigest()

### … With Considerations

Some parameters should partition/collate output, not updating an opaque data version

SALT\_IGNORE = set()

**class** **AggregateArtists**:

...

SALT\_IGNORE.add(date\_interval)

**def** **output**(self):

...

**return** LocalTarget(f'{self.date\_interval}/aggregate-{salt}.parquet')

**def** **get\_salted\_version**(task):

...

msg += [

'{}={}'.format(

param\_name,

repr(task.param\_kwargs[param\_name]))

**for** param\_name, param **in** sorted(task.get\_params())

**if** param.significant

**and** param **not** **in** SALT\_IGNORE

]

...

This works when the param is in the file name, but if you switch the value in an

input to a new task, you need to bump that task’s version!

## The Signature Input

If your task is simple enough, you may have luck with a signature input. This input should be rich enough to expose any nuance, feature, or bug. Call task on input, and hash output!

**class** **AutoVersion**:

**def** **\_\_init\_\_**(self, signature\_input, base=''):

...

**def** **\_\_get\_\_**(self, task, owner):

h = sha256(self.base)

h.update(

task.process(self.signature\_input())

)

**return** h.hexdigest()

**class** **AddTask**(Task):

\_\_version\_\_ = AutoVersion(**lambda**: **0**)

value = IntParameter()

**def** **process**(self, data):

**return** add(data, self.value)

**def** **run**(self):

**with** self.output().open('w') **as** f:

f.write(self.process(self.input()))

Use a real-ish input to an aggregation task

**class** **Aggregate**(Task):

\_\_version\_\_ = AutoVersion(

**lambda** : DataFrame(

{

'col1':[**1**, **2**],

'col2':[**2**, **3**],

}

)

)

Two variants of process should result in new \_\_version\_\_’s! … you just need to implement a hash for a DataFrame!

**def** **process**(self, data):

**return** data.groupby('col1').sum()

**def** **process**(self, data):

**return** data.groupby('col2').sum()

## Custom Salts

|  |  |
| --- | --- |
|  | Add new class pram w/o chnaging salt |

## Data-Driven Salts

|  |  |  |
| --- | --- | --- |
| Base signature on query time… data will automatically refresh |  | |
|  | | Use the actual data to sign the output!! Many big data stores (HBase) store data versions like this automatically. Use this paradigm to kick off Big Data pipelines only when the data has changed! |

# LUIGI

## Atomicity

**Ideally**: an indivisible and irreducible series of operations such that either all occur, or nothing occurs 🡺 no partial data files left over

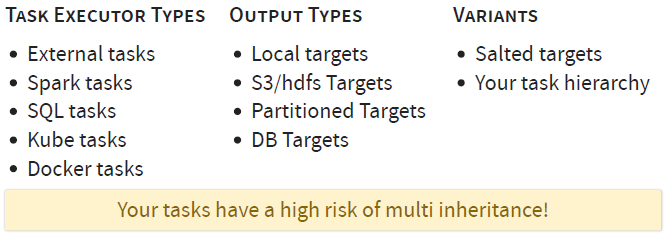
* Define .write(), .exists() atomically for a target
  + May never return True for an incomplete or failed output
* A task is runnable when all input targets .exists()
* A task is done when it’s output target .exists()

**Algorithm**: (a) Write to tempfile (b) Rename or Move on success (atomic)

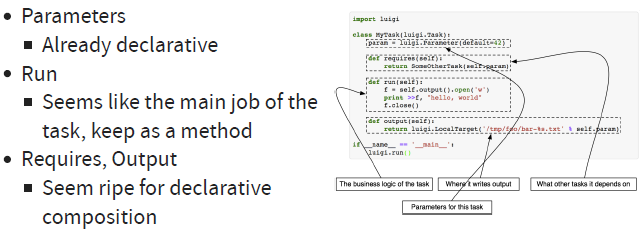
|  |  |
| --- | --- |
|  | Tasks .require() other Tasks, forming a DAG (they don’t require Targets!) |

## Tasks





### Task Skeleton



**class** **AggregateArtists**(Task):

# Declarative Params, composition

date\_interval = DateIntervalParameter()

**def** **output**(self):

**return** LocalTarget("data/artist\_streams\_{}.tsv".format(

self.date\_interval))

**def** **requires**(self):

**return** [Streams(date)

**for** date **in** self.date\_interval]

**def** **run**(self):

artist\_count = defaultdict(int)

**for** input **in** self.input():

**with** input.open('r') **as** in\_file:

**for** line **in** in\_file:

ts, artist, tr = line.split()

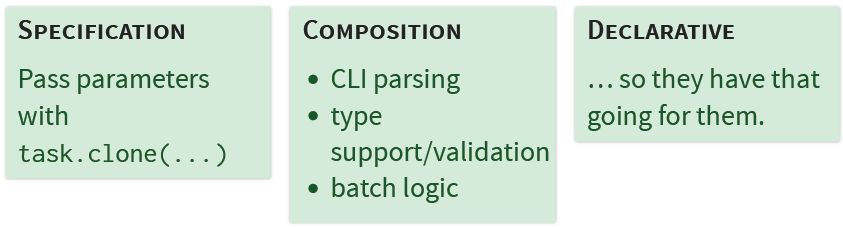
artist\_count[artist] += **1**

**with** self.output().open('w') **as** out\_file:

**for** artist, count **in** artist\_count.items():

out\_file.write(artist, count)

## Parameters



|  |  |  |
| --- | --- | --- |
| *Params primarily implement type conformity and serialization*  **class** **Parameter**:  **def** **parse**(self, x): # from str  **def** **serialize**(self, x): # to str  **def** **next\_in\_enumeration**(self, \_value) |  | |
| **class** **Task**:  **def** **\_\_init\_\_**(self, \*a, \*\*k):  params = self.get\_params()  param\_values = self.get\_param\_values(params, a, k)  # Set all values on class instance  **for** key, value **in** param\_values:  setattr(self, key, value)  **def** **get\_param\_values**(self, params, a, k):  ...  **for** param\_name, arg **in** six.iteritems(k):  result[param\_name] = params\_dict[param\_name  ].normalize(arg)  ... | | *Here, the task appropriately delegates initialization and serialization to the params.* |

## Atomic Targets

**target.exists()** – returns true iff target successfully written (if atomic write, just check for file existence

**target.open()** – atomic writer for clean states and to satisfy .exist()

iff **task.output().exists()** – task is done

Task ready to run iif:

all([

req.exists()

**for** req **in** flatten(task.requires())

])

## Task Types

|  |  |
| --- | --- |
| **EXTERNAL**  **class** **RawData**(ExternalTask)  # Wraps existing data  # Not managed by Luigi  # No .run() function  **def** **output**(self):  ...  These are (usually) your real inputs | **WRAPPERS**  **class** **AllReports**(WrapperTask)  # Organize other tasks  # No .run() or .output()  **def** **requires**(self):  **return** map(  self.clone,[Task1,Task2]  )  ...  These are good surrogate outputs |
| **EXTERNAL PROGRAM**  # luigi.contrib.external\_program  **class** **ExternalProgramTask**(Task):  **def** **program\_args**(self):  **raise** **NotImplementedError**()  **def** **run**(self):  args = self.program\_args()  logger.info(  'Running command: %s',  ' '.join(args))  subprocess.check\_call(args) | # luigi.contrib.spark  **class** **SparkSubmitTask**  (ExternalProgramTask):  **def** **program\_args**(self):  **return** [  'spark-submit',  '--name', self.name,  self.app  ...  ]  **class** **PySparkSubmitTask**  (SparkSubmitTask):  ... |

### External Programs

* No need to implement in python! Use ExternalProgramTask to kick of a job via the command line in any language/framework you choose
* Your luigi target doesn’t control how an external program writes data - it can only point to the data once written.
* Choose a target that matches the atomicity pattern of the program (Eg, luigi.contrib.S3FlagTarget which looks for a \_SUCCESS flag left by many hadoop/EMR jobs, including spark
* Two flavors: (a) **Local/Wrapped Runner**: The job process starts and locally succeeds or fails. Examples: python main.py, spark --deploy-mode client (b) **Dispatched Runner**: Process submits a remote job and successfully terminates. The task output could be job id; next task queries job status (out of scope for this class). Examples: Oozie, EMR step, spark --deploy-mode cluster

## Output

Rules to choose from:

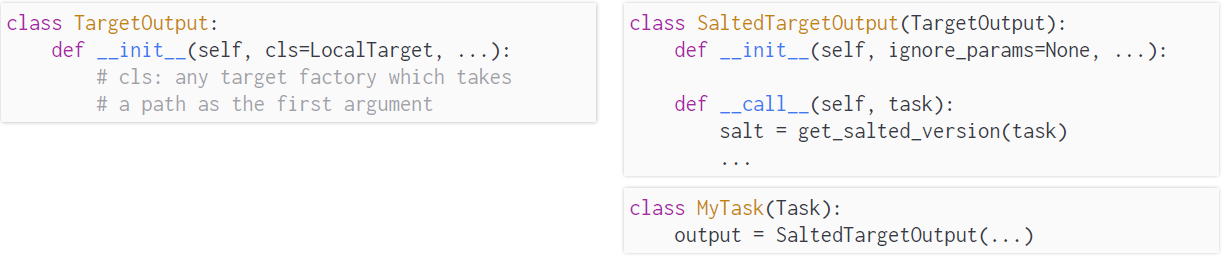
(1) Any relationship between file path & task name? eg, w/ configurable root directory {DATA\_ROOT}/{param}/{task\_name}.txt (2) Is output salted?

(3) Defaults (e.g. encrypting)? (4) Output local, S3 targets ..? Authentication?

Output logic unrelated to task classes 🢥 Composition with descriptors

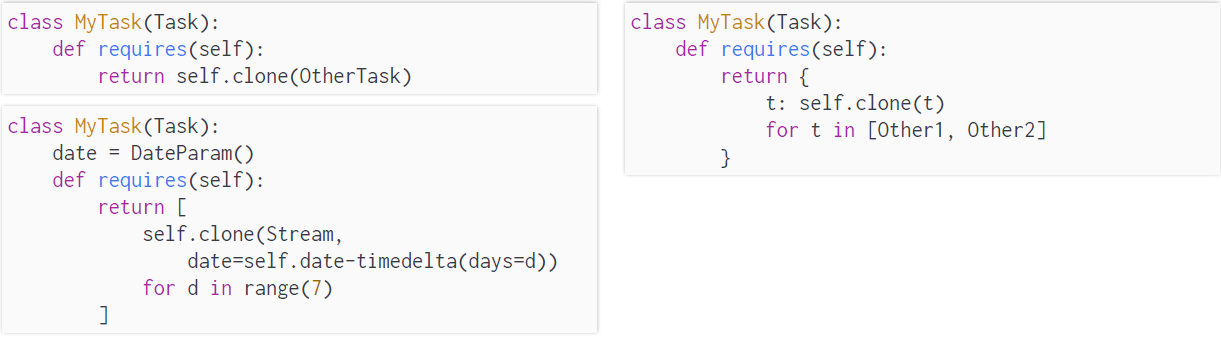
|  |  |
| --- | --- |
| **class** LocalTargetOutput:  **def** \_\_init\_\_(self,  file\_pattern='{task}',  ext='.csv'):  ...  **def** \_\_get\_\_(self, task, cls):  *# Make it look like a method*  **return** **lambda**: self.output(task) | **class** MyTask:  *# Compose output*  output = LocalTargetOutput('{task.date}/ {task.\_\_class\_\_.\_\_name\_\_}')  >>> MyTask().output()  LocalTarget('2019-01-01/MyTask.csv') |
| **def** \_\_call\_\_(self, task):  **return** LocalTarget(  self.file\_pattern.  format(task=task)  + self.ext) | Pattern \_\_get\_\_ makes MyTask().output return callable, since Luigi expects a method function, wants to say MyTask.output() |

… and extend functionality with min. effort



## Requirements

🞟Can be: single task, list, dict 🞟Share params with task 🞟Be formed with task.clone(requirement\_class, \*\*param\_overrides)



🢥 Composition with descriptors 🞟Give nice type completion (self.other) + Make class more declarative

|  |  |
| --- | --- |
| **class** Requirement:  **def** \_\_init\_\_(self, task\_class, \*\*params):  ...  **def** \_\_get\_\_(self, task, cls):  **return** task.clone(self.task\_class,  \*\*self.params) | **class** MyTask(Task):  *# Replace task.requires()*  requires = Requires()  other = Requirement(OtherTask)  **def** run(self):  *# Convenient access here...*  **with** self.other.output()  .open('r') **as** f:  ...  >>> MyTask().requires()  {'other': OtherTask()} |
| **class** Requires:  **def** \_\_get\_\_(self, task, cls):  **return** **lambda** : self(task)  **def** \_\_call\_\_(self, task):  *# Search task.\_\_class\_\_ for Requirements*  *# return instances* |

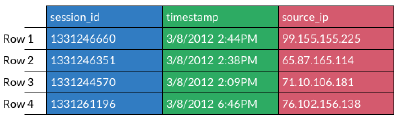
Two compositional tools in Luigi Utils – Work for basic param sharing

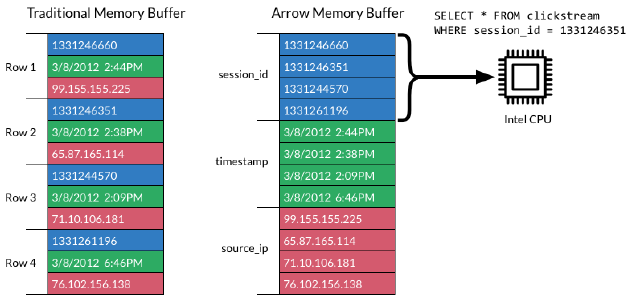
|  |  |
| --- | --- |
| *# Copies parameters*  @inherits(TaskA)  **class** TaskB:  ... | *# Copies params and requires*  @requires(TaskB)  **class** TaskC(Task):  ... |

BUT force inheritance of unnecessary prameters🢥More flexible approach:

|  |  |
| --- | --- |
| **class** TaskA(Task):  requires = Requires()  *# Type completion for forks*  in1 = Requirement(Task1)  in2 = Requirement(Task2)  **class** TaskA2(TaskA):  *# Auto inherits in1, in2*  in3 = Requirement(Task3) | **class** ComplexRequirement(Requirement):  **def** \_\_init\_\_(self, factory):  self.factory = factory  **def** \_\_get\_\_(self, task, owner):  **return** self.factory(task)  **class** TaskC(Task):  data = ComplexRequirement(  **lambda** task: task.clone(OtherTask,  param=translate[task.param])) |
| **class** NeighborhoodTask(Task):  neighbors = NeighorhoodRequirements(Task1)  **class** RollingAggregation(Task):  history = HistoricalRequirements(Task2, days=7)  **class** HistoricalRequirements (Requirement):  **def** \_\_init\_\_(self, task\_class, days=7):  ...  **def** \_\_get\_\_(self, task, owner):  ...  **return** [task.clone(  self.task\_class,  date=task.date - timedelta(days=d))  **for** d **in** range(self.days)] | **class** NeighborRequirements(Requirement):  **def** \_\_get\_\_(self, task, owner):  neighbors = {}  **for** yn, yo **in** {'n':1,  'c':0, 's':-1}:  **for** xn, xo **in** {'e':1,  'c':0, 'w':-1}:  neighbors[yn+xn] = task.clone(  self.task\_class,  x=task.x + xo,  y=task.y + yo,)  **return** neighbors  We can write functions that create a collection of requirements seeded from a single taks,, eg used for historical range lookups or task-based convolutions |

# APACHE PARQUET/ARROW



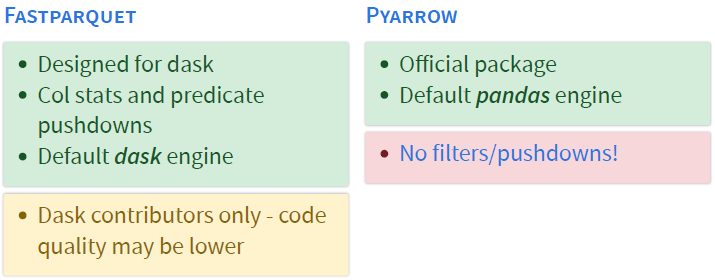


|  |  |
| --- | --- |
| A picture containing screenshot  Description automatically generated | **Apache Parquet**  Emerging standard for distributed,  columnar binary data Statistics for predicate pushdown  <http://parquet.apache.org/>  **Statistics**  Parquet stores partition  min/max values to allow for  index and block filtering too! |

## Column Stores

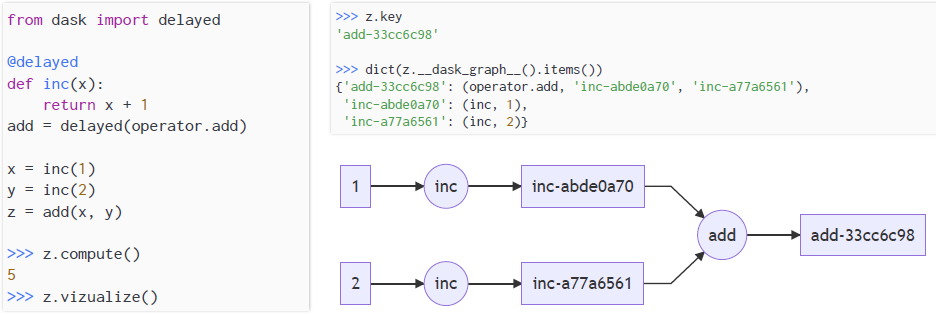
|  |  |
| --- | --- |
| A screenshot of a cell phone  Description automatically generated | Most operations only care about a few columns. Don’t read them all if you don’t need to! |
| A screenshot of a cell phone  Description automatically generated | Column stores offer superior:   * Compression, due to homogeneity * Selective column reads * IO, due to buffering |
| Key-Value Stores Fast lookup/ range scans on key, but not other values. Values are arrays, unstructured, dicts Key ranges possibly partitioned Column Stats Parquet stores statistics about each row group: min/max vals, n unique vals, which row groups/parts exist optionally |  |

## FASTPARQUET V. PYARROW



# DASK

**DASK provides:** 🞟Symbolic: symbolic, delayed, computational graph API 🞟Cluster : distributed computation of graph 🞟Wrappers Hi-level wrappers of numpy, pandas, etc on top of Dask graphs. **DASK is**: computational graph built on ***delayed*** 🢥 Turns functions into objects; when called, return DAG encoded in K/V (key/value) store: delayed object’s + value = tuple (function to call + args = keys in graph to other delayed results). If key in graph instead of coded value, Dask recursively evaluate key, store intermediate result then calculate target result. This is handled by dask scheduler or executor. Dask keys = combination of function names, hashes of input, and UUID’s generated at runtime to guarantee globally unique lineages, but not stable enough for persisted output like salted graphs.



## Result in Pieces

|  |  |
| --- | --- |
| df.groupby('col').sum()  🡙  sum(  map(  lambda d: d.groupby('col').sum(),  partition(df, n\_partitions)  )  ) | # Outer blocks  BR = mtx\_a.shape[0]  BC = mtx\_b.shape[1]  INNER\_BLOCKS = mtx\_a.shape[1]  # == mtx\_b.shape[0]  # "Outer" matrix:  # each part sums delayed dots  # of the inner  dot = delayed(np.dot) |
| def delayed\_dot(mtx\_a, mtx\_b):  """Delayed block-wise dotproduct. Each input is  'matrix of matrices', eg::  mtx\_a[r, c] = delayed(ndarray(10, 10))  """ | out = np.asarray([  [  sum([  dot(a, b)  for a, b in zip(  mtx\_a[block\_r, :],  mtx\_b[:, block\_c]  )  ]) for block\_c in range(BC)  ] for block\_r in range(BR)  ])  return out |

## Split, Apply, Combine

df.groupby().sum()

|  |  |
| --- | --- |
|  | **vanilla groupby:** split on grouping column, apply = reduction, combine = concatenation.  **dask** occur inside each apply. Dask arrays,  dataframes split by ‘block’ & ‘partition’ resp. Operations considered apply of partitions, combine step collate results into one or more output partitions. |

## Dask.DataFrame

|  |  |
| --- | --- |
| dask has not read the data yet. It did brief checking to get row headers and column types, but data not yet in memory. | A screenshot of a cell phone  Description automatically generated  **predicate pushdown**: This row index must fall within a  single partition, so dask only returns that. **Not always** exact (use to exclude chunks, but may get rows that don’t match). Use as an optimization technique, not a query: read\_parquet(path, filters=[('a', '=', 5)]).query("a == 5") |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Predicates | Dask exposes predicate pushdown in two ways  **Predicate pushdown** = Optimization to apply conditions (predicates) early to prevent loading of unneeded rows. In stead of:  SELECT c1, c2, ...  FROM (SELECT c1, c2, ...  FROM base\_table  ) derived\_table  WHERE c1 = ?  ⇨ PUSHDOWN:  SELECT c1, c2, ...  FROM (SELECT c1, c2, ...  FROM base\_table  WHERE c1 = ?  ) derived\_table | | | |  | |
| Performance boost using index divisions is massive. Basic aggregations are fine. If task involves a WHERE, GROUP BY? Use sorted index column🢥DataFrame more like a Key/Value store! Use the Index | | | df.divisions are the index bounds of each dask partition |  | | |
| A screenshot of a cell phone  Description automatically generated | | A screenshot of a cell phone  Description automatically generated  Note the \_meta. Dask must know shape of delayed data; can’t always guess. Provide when you can | | | |  |

### Indexed DataFrames – Surrogate keys

|  |  |  |
| --- | --- | --- |
|  |  | In vanilla Pandas, we can have a multi-column index which is jointly sorted. Dask doesn’t support this, but standard practice in Big Data K/V stores is to design a **surrogate index**, a function of the columns you want to search by, which is optimized for your lookup patterns |
|  |  | This dask/parquet dataset is fast to scan students for a given semester. We can  look up a single student quickly given semester. May need full scan if semester unknown. |

**Fancy indexing** passes array of indices to access multiple array elements at once

**Parquet** (and other dask collections) look for folders, not files. They implement their own file protocols, and don’t want to use **luigi’s**

## Dask Debugging

Dask debugging difficult for 2 reasons: (1) Asynchronous execution stepping through/ debugging tough (2) Confusion when error raised (“Issue writing to parquet…”)

### Synchronous Debugging

Use single-threaded scheduler:

>>> x.compute(scheduler='single-threaded')

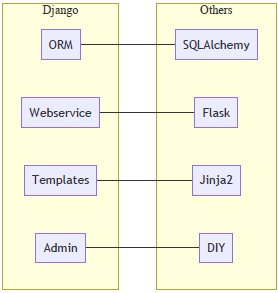
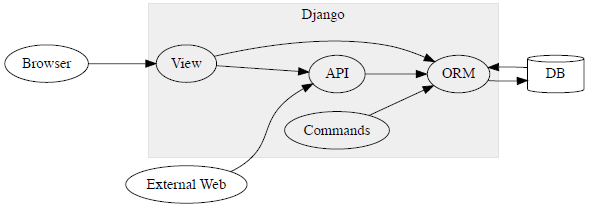
**with** dask.config.set(scheduler='single-threaded'):

run\_some\_code()

**Delayed Confusion**: Because dask is lazy, it won’t raise errors until you compute

# DJANGO

Django is an entire application framework for webapps and more



## 2 Types of Data Use

|  |  |
| --- | --- |
| **Analytics**  summary stats, care about specific  columns, aggregations, etc | **Objects**  want all of the data per row, and often save new rows |
| **Analytics – Data Mapper** | **Objects – Active Record** |

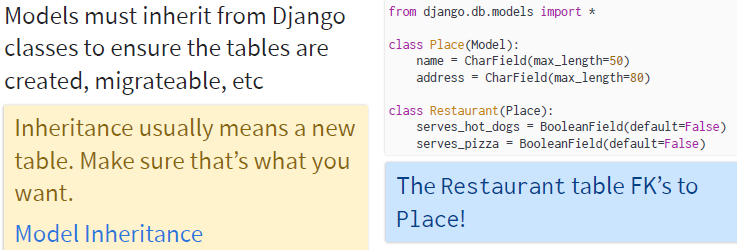
## ORM (Object Relational Mapping)

|  |  |
| --- | --- |
|  | 🞟.order\_by('-col') means ‘descending order’ of col  🞟 some\_query[0] or some\_query[:5] is an sql LIMIT, not list access |

In memory SQLite DB’s for Unittest



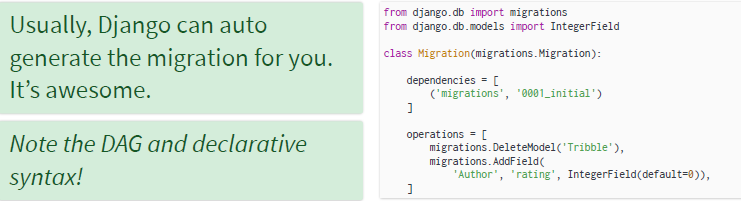
## Inheritable Models



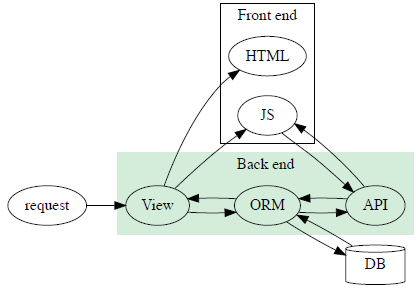
## Composable Models

|  |  |  |  |
| --- | --- | --- | --- |
| **Django models: great use of Composability** | | | Manager objects:  composable ‘views’ |
|  | Subclass Manager, QuerySet to reuse functionality across unrelated models in composeable way (solution adapted from <https://gist.github.com/ryanpitts/1304725>) **Note**: '\_\_' syntax in Django filters implies a lookup, eg:  “”” Annotate each grade by looking up the timestamp of all other grades # from the same student “””  Grades.objects.annotate(mx=Max('student\_\_grades\_\_tstamp')) | |

## Migrations



## Page Lifecycle



## Django Views

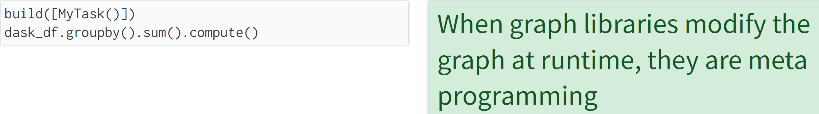
|  |  |
| --- | --- |
| **URL Routing**  **from** **django.urls** **import** path  **from** **.** **import** views  urlpatterns = [path('now', views.current\_datetime),path('articles/<int:poll\_id>/', views.detail),] | **The View**  **def** current\_datetime(request):  now = datetime.datetime.now()  html = """  <html><body>It is now {}.</body></html>  """.format(now)  **return** HttpResponse(html)  **def** detail(request, poll\_id):  **try**:  p = Poll.objects.get(pk=poll\_id)  **except** Poll.DoesNotExist:  **raise** Http404("Poll does not exist")  **return** render(request, 'polls/detail.html', {'poll': p},) |

# MetaProgramming

**Metaprogramming** = technique for programs to treat programs as their data ⇨ program designed to read, generate, analyse, transform programs, even modify itself while running

A screenshot of a cell phone

Description automatically generated



A screenshot of a cell phone

Description automatically generated

## MetaClasses

A screenshot of a social media post

Description automatically generated

A screenshot of a social media post

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a social media post

Description automatically generated

You don’t need metaclasses (until you do). But they are critical to understand complex code

**The Point**: Metaprogramming allows you write **Domain Specific Language**, big

or small ⇨ Isolate logic (metaprogram) from data (actual program)

A screenshot of a cell phone

Description automatically generated

## Model Options

A screenshot of a cell phone

Description automatically generated

## ModelBase

A screenshot of a cell phone

Description automatically generated

# API

## ORM as a Service

A django view hard-bakes the data into the page You want to expose your data directly to clients, whether for the web via JS, or any other service (even python)

## Django REST Framework

It’s not really REST, but that’s okay. It emphasizes multiple linked endpoints for ‘raw’ data access It emphasizes development and debugability

## Serializers

|  |  |  |
| --- | --- | --- |
| It’s easy to expose any Django model… | **class** AccountSerializer(ModelSerializer):  **class** Meta:  model = Account  fields = '\_\_all\_\_' | |
| **class** CommentSerializer(serializers.Serializer):  email = serializers.EmailField()  content = serializers.CharField(max\_length=200)  created = serializers.DateTimeField() | | .. or any othe object |

## Views

|  |  |
| --- | --- |
| **class** SnippetList(APIView):  **def** get(self, request, format=None):  snippets = Snippet.objects.all()  serializer = SnippetSerializer(snippets, many=True)  **return** Response(serializer.data)  **def** post(self, request, format=None):  serializer = SnippetSerializer(data=request.data)  **if** serializer.is\_valid():  serializer.save()  **return** Response(serializer.data,status=status.HTTP\_201\_CREATED)  **return** Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST) | … and pack it all into a view |

## Django REST Witchcraft

django-rest-witchcraft = extension for Django REST Framework that adds support for SQLAlchemy. Aims to provide similar development experience to building REST api’s with Django REST Framework with Django ORM, except with SQLAlchemy

## Swagger

*The OpenAPI Specification … is the world’s standard for defining RESTful interfaces. The OAS enables developers to design a technologyagnostic API interface that forms the basis of their API development and consumption*. <Swagger.io> API is data (generated from backend or vice versa). Metacode for client libraries provides consistency + removes boiler plate. Enables new workflows: 🞟DRF -> generate API spec -> generate client 🞟Write spec for existing API -> generate client 🞟Generate server stubs from spec, implement them

## API and Clients

**API**: 🞟simple, structured way of interacting with your program 🞟Way for microservices to communicate 🞟For serving data via ORM… 🞟… or configuring your program! 🞟… or returning a model prediction

### HTTP Methods

🞟 **CRUD** - Create, Retrieve, Update, Delete 🞟GET: Retrieve information 🞟POST: Take an action (not idempotent) 🞟PUT: Store or modify (eg put) data (idempotent) 🞟DELETE: Remove a resource <https://stackoverflow.com/questions/630453/put-vs-post-in-rest> No need to support both PUT and POST. Depends on what object referencing in request. Considerations: 1) Do you name your URL objects you create explicitly (PUT), or let the server decide (POST)? 2) PUT is idempotent, so if PUT an object twice, it has no effect 🢥 use PUT when possible. 3) Update/create resource with PUT with the same object URL 4) With POST can have 2 requests at the same time making modifications to a URL, update different parts of the object.

## API Views

|  |  |
| --- | --- |
| **class** SnippetList(APIView):  **def** get(self, request, format=None):  snippets = Snippet.objects.all()  serializer = SnippetSerializer(snippets, many=True)  **return** Response(serializer.data)  **def** post(self, request, format=None):  serializer = SnippetSerializer(data=request.data)  **if** serializer.is\_valid():  serializer.save()  **return** Response(serializer.data,status= status.HTTP\_201\_CREATED)  **return** Response(serializer.errors, status = status.HTTP\_400\_BAD\_REQUEST) | DRF makes it easy to  implement related  methods |
| **class** MLModelView(APIView):  **def** \_\_init\_\_(self, pickle\_file):  self.model = open(pickle\_file).read()  **def** get(self, request, format=None):  ser = SklearnSerializer(self.model)  **return** Response(ser.data)  **def** post(self, request, format=None):  x = PredictionXSerializer(data=request.data)  **if** x.is\_valid():  y = PredictionYSerializer(self.model.predict(x.save()))  **return** Response(y.data)) | … whether for ORM  or other data |

## Client: Requests

DRF primarily deals with serving data. We must still connect to a server when we want data as a client!

|  |  |
| --- | --- |
| >>> r = requests.get(  'https://api.github.com/user',  auth=('user', 'pass'))  >>> r.status\_code  200  >>> r.headers['content-type']  'application/json; charset=utf8'  >>> r.encoding  'utf-8'  >>> r.json()  { 'private\_gists': 419,  'total\_private\_repos': 77,...} | A best-of-breed generic library requests - Use whenever you want to write you own python client against any API - Requests allows you to send organic, grass-fed HTTP/1.1 requests, without the need for manual labor. There’s no need to manually add  query strings to your URLs, or to form-encode your POST data. Keep-alive and HTTP connection pooling are 100% automatic, thanks to urllib3 |

## DRF CoreAPI

|  |  |  |
| --- | --- | --- |
| **from** **rest\_framework.documentation** **import**  include\_docs\_urls  urlpatterns = [  ...  url(r'^docs/',  include\_docs\_urls(title='My API title'),)]  Endpoints:  /docs/  /docs/schema.js *# Programatic Specification* | | By Default, Django Rest Framework exposes a ‘CoreAPI’ specification Schema generation is what  makes this self documenting |
| **CLI**  $ pip install coreapi-cli  $ coreapi get http://127.0.0.1:8000/  <Pastebin API "http://127.0.0.1:8000/">  users: {  list([page])  retrieve(pk)  }  **USAGE**  $ coreapi action users list  [  {  "url": "http://127.0.0.1:8000/users/2/",  "id": 2,  "username": "aziz",  "snippets": []  },  ...  ] | Specification is sufficient to generate an automatic client! Core API is a format-independent **Document Object Model** for representing Web APIs. Used to represent either Schema or Hypermedia responses, and allows you to interact with an API at the layer of  an application interface, rather than a network interface. Available for Core JSON, Open API/ Swagger, HAL, and JSON Hyper-Schema. Command line tool to interact with APIs exposing any of these formats, as well as a Python client library. Using a Core API client is a more robust and meaningful way to interact with your API than constructing HTTP requests and decoding responses. The dynamic client library is always up to date with the API, and client code focuses solely on the interface being provided, rather that dealing with network details and encodings. | |
| **PYTHON**  $ pip install coreapi  **import** **coreapi**  client = coreapi.Client()  schema = client.get('https://api.example.org/')  **USAGE**  >>> client.action(schema, ['users', 'list'])  [  {  "url": "http://127.0.0.1:8000/users/2/",  "id": 2,  "username": "aziz",  "snippets": []  },  ...  ] | We’ll capture some amount of ‘best practices’ via using a consistent mechanism to hit the API, but all the interactions are still basically data (eg the actions are just strings, and the responses are json/dicts/lists instead of native objects of the type handled by the API server side). This limits the value vs using requests directly, though it is still likely better than doing everything yourself.  **The Point**: 🞟API specifications allow self-documentation as well as auto-generating  client library code 🞟Enable full-cycle ‘dogfooding’: test your server with your generated client, ensuring the API spec itself is accurate! 🞟Even if you want to use an API that doesn’t publish a specification, consider writing one instead of writing the API client from scratch | |

# Mutable Objects

|  |  |
| --- | --- |
| **Mutable**  [] = List  {} = Dict  set()  d = Task(); d.a = 134 #Instance mutable | **Immutable**  a = (1, 2) # Tuple  frozenset  namedtuple  # frozendict  >>> a[0] = 1 (TypeError: 'tuple' object not support item assignmt)  >>> a.x = 1 (AttributeError: 'tuple' object has no attribute 'x') |
| >>> a = [1, 2]  >>> o = a  >>> o **is** a  True  >>> a += [3]  >>> a  [1, 2, 3]  >>> o **is** a  True | >>> a = (1, 2)  >>> o = a  >>> a += (3,)  >>> a  (1, 2, 3)  >>> o **is** a  False *# Same if a is an int* |

The += operator assigns a new tuple object to the label a, as opposed

to modifying the list labelled a in place. Special:

*# Example: collection of (x, y) points*

>>> origin = (0, 0)

>>> x = (1, 0)

>>> y = (0, 1)

>>> points = set([origin, x, y])

>>> (0, 0) **in** points

True

>>> [0, 0] **in** points

**TypeError**: unhashable type: 'list'

## Pass By Ref or Value

|  |  |
| --- | --- |
| X = 5  O = x  X +=1  Assignment points a label to a value. Python is **pass by value** where the values are labels! |  |
| a = [1, 2]  a.append(3)  Only mutable objects allow the  actual data to change! (typically  through methods) |  |

## Mutability vs State

|  |  |
| --- | --- |
| **Mutable**  An object/system which can change (even if it never does). Immutable objects can be more performant | **Stateful**  An object/system which has a state, which may be mutable or not. State can impact execution, even if state immutable. Stateless implies immutable! |

Everything is a mutable dict! Mutable types make for easy coding, at some overhead cost. Sometimes it is worth working with tuples for performance. DB queries on moderate to large data: consider data type/ serialization mechanics. By default, Django queries objects back in a json-like form, and the returned rows are dictionaries. This makes it safe and debuggable to retrieve results, turn them into ORM objects, etc. However, for a columnar analytics workflow (with known schema or columns), instruct Django to iterate through rows as a tuple given a fixed set of columns (qs.values(\*cols) vs qs.values\_list(\*cols)). This reduces size of the DB payload by avoiding returning column names in every record, and streamlines the python iteration and memory footprint. It’s a no-brainer if turn query into a DataFrame in python, since you can reconstruct the columns using the same schema you requested!

### Immutable Workflows

🞟A Luigi workflow implies immutable artifacts 🞟Immutable data output is a Very Good Thing 🞟It provides reproducibility and repeatability 🞟Never ‘just edit the input’. You need a makefile, or .. 🞟You can, of course, delete outputs and run again 🞟Salted Graphs provide a compromise: keep the immutable workflow, change version/target when output is expected to be different

### Deterministic Workflows

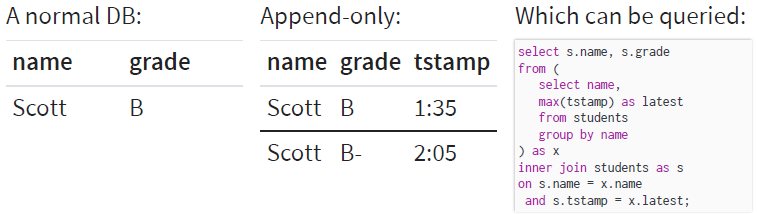
🞟Immutable data, version control, and data lineage 🞟(Salted) implies 100% deterministic output 🞟If we have the code and the input data, we will get the same results every time 🞟Unless, of course, you use a random number

## Not Every Workflow is Deterministic

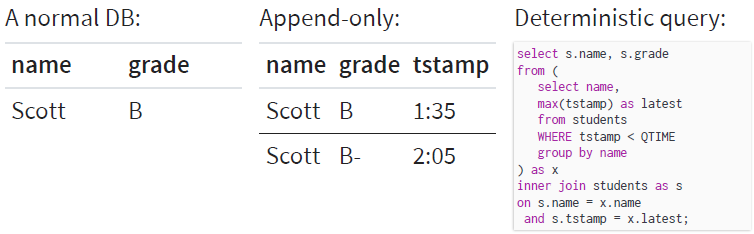
|  |  |
| --- | --- |
| **Weak Mutability** | **… Minibatch** |
|  | **… Input Snapshots** |

## Snapshots

Normally, ‘name’ = unique key on student grade table. We can have a unique key of (name, tstamp) and allow inserting updates to grade at any time. We simply must query for the rows that have the maximum timestamp per student



Make view deterministic? impose query/snapshot time other than now. Can filter view to only include timestamps below that query time when calculating max timestamp per student



|  |  |
| --- | --- |
| Deterministic query time is not now()!  Consider an incremental backoff depending on when you expected data to have changed Here, we update the query every hour for the first day, then every day for the first week, then every subsequent Monday  << SEE [CUSTOM SALT](#_Custom_Salts)>> |  |

# Atomicity

## Atomic Writes

… ensuring your files are full and complete and (ideally) idempotent

# This the BAD way

big-slow-calculation > /outputs/foo.data

# This is the good way

big-slow-calculation > /outputs/foo-tmp-**123456.**data

mv /outputs/foo-tmp-**123456.**data /outputs/foo.data

|  |  |
| --- | --- |
| **Transactional**  **with** transaction.atomic():  *# NB: this only for the default DB!*  MyModel.objects.create(...)  SQL gives us clean all-or-nothing  transactions (usually) | **Workflow**  **class** DjangoModelTarget(Target):  **def** exists(self):  *# Check if row exists??*  Luigi wants a target to complete Atomically |

# Database

## Luigi Targets

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *# luigi.contrib.postgres*  **class** PostgresTarget(Target):  **def** \_\_init\_\_(self, update\_id, \*\*conn\_keywords):  ...  **def** exists(self):  *# Check \*metadata\*: query run?*  "SELECT 1 from {marker\_table}"  " where update\_id={self.update\_id}"  Luigi has builtin sqlalchemy Target’s, Query tasks for Postgres, Redshift, gen RDBMS | | | update\_id can be used to implement a salted workflow! Relying on an ‘existence’ table violates a pure ‘can I read this target’ form of atomicity, Can tasks modify rows, or just append, or must they support both? What about unique keys? | |
| **class** DjangoModelTarget(Target):  **def** \_\_init\_\_(self, model, \*\*unique): ...  **def** get(self):  *# Assumes unique constraint*  **return** self.model.objects.get(\*\*self.unique\_kwargs)  **def** exist(self):  **try**:  m = self.get()  **return** True  **except** self.model.DoesNotExist:  **return** False | | | | You can easily write a target to check for existence of a unique (or unique\_together) key |
| You can easily write a target to check for existence of a unique (or unique\_together) key. You can even support salted targets | **class** SaltedDjangoModelTarget(Target):  **def** \_\_init\_\_(self, model, version, \*\*unique): ...  **def** get(self):  *# Assumes there is a unique constraint*  **return** self.model.objects.get(version = self.version, \*\*self.unique\_kwargs) | | | |
| **class** MultiTarget(DjangoTarget):  **def** exists(self):  **return** self.model.filter( \*\*self.kwargs).count()>0  **with** atomic():  target.model.bulk\_create(...) | | If you can exhaustively partition the rows between multiple targets, we can continue the pattern to multi-row, non-unique Inserts. Except if a valid task should insert 0 rows, eg an empty dataset, and you want to know that it should be empty! | | |

## Normalization

Normalization entails organizing the columns (attributes) and tables (relations) of a database to ensure that their dependencies are properly enforced by database integrity constraints

|  |  |
| --- | --- |
| **Denormalized**  **class** Grade(Model):  student = CharField()  assignment = CharField()  grade = FloatField()  Grade.objects.values('student'). annotate(grade=Avg('grade')) | **Normalized**  **class** Assignment(Model):  assignment = CharField()  **class** Student(Model):  name = CharField()  **class** Grade(Model):  student = ForeignKey(Student)  assignment = ForeignKey(Assignment)  grade = FloatField() |
| 3 options to query the Grade table:  1. **Grouping by student** results in an ORM Student object being returned (pythonic), but must run a SQL join in the backend and return all data related to the student. Least performant option.  2. **Group by unique id** that FK’s to the student table. Fastest since no join is needed, but results in a private key representing the student being returned. Bad form: DBA can change meaning of the private id at any time. Often safe enough so long as that private key is not persisted.  3. **Group by what believed to be unique** on the Student table. A join is still done but only return necessary data to calculate the unique grade. If the table is unique on a combination of fields (ie first and last names this gets much trickier). | **Normalized (how to query)**  **def** vl(\*args):  **return** Grade.objects.values\_list(\*args)  *# What you'd expect*  >>> vl('student').annotate (grade=Avg('grade'))  [(Student('scott'), 89), ...]  *# Fast and minimal, avoid join*  >>> vl('student\_id').annotate(...)  [(1, 89), ...]  *# Join on 'real' value*  >>> vl('student\_\_name').annotate(...)  [('scott', 89), ...] |

## Stars and Flakes

|  |  |
| --- | --- |
| **Fact Tables**  **class** SalesFact(Model):  *# NB: Not DateField!*  date = ForeignKey(DateDim)  product = ForeignKey(ProductDim)  *# Facts (theory: just 1)*  units = IntField()  revenue = FloatField()  total\_sales = SalesFact.objects.sum() | **Dim Tables**  **class** DateDim(Model):  date = DateField()  year = IntField()  season = CharField()  quarter = IntField()  is\_holiday = BooleanField()  Dims may store derived fields! Compare:  SalesFact.objects.filter(  date\_\_year=2018, *# Join and lookup, vs*  date\_\_date\_\_year=2018 *# sql-side YEAR(date)*  ).sum() |

SalesFact.objects.filter(date\_\_year=2018) says “for every sales row, join the date field to the row in DateDim and see if the year column there is 2018”.

.filter(date\_\_date\_\_year=2018) says “do the join, then call YEAR(date) on the date column of the DateDim table to see if that is 2018”.

If SalesFact had date = DateField instead of date = ForeignKey(DimDate) then .filter(date\_\_year=2018) would be the equivalent of:

select \* from sales\_fact where YEAR(date) = 2018

because field\_\_something in the filter is a registered lookup function on the field; lookups across FK’s are a join and point to the joined table, whereas \_\_year, \_\_month, etc are registered lookups for date fields that calculate those properties. There are many other lookup functions and you can write your own as well.

**Star schemas** are highly denormalized and usually don’t guarantee unique dims! Avoid querying dim’s by PK directly! **Snowflake**: Fully normalized dimensional variant of Star

More referential integrity, but harder to query

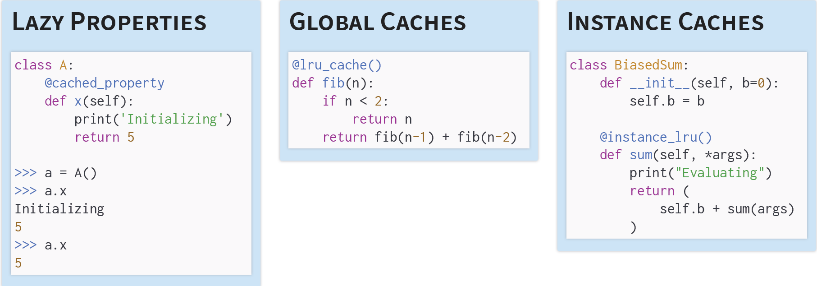
## Intelligentg Keys

|  |  |
| --- | --- |
| The primary key on a table is usually auto-incrementing and best considered ‘private’  But, Eng teams (and you) can optimize it. Beware over-optimization! This can kill readability … or can be a critical performance boost, eg for partitions by date. SQL can’t optimize queries across the join | **class** DateDim(Model):  date = DateField()  **def** save(\*\*kwargs):  **if** **not** self.pk:  self.pk = self.get\_pk(self.date)  super(DateDim, self).save(\*\*kwargs)  @classmethod  **def** get\_pk(cls, date):  **return** int(date.strftime('%Y%m**%d**'))  SalesFact.objects.filter(date\_id=DateDim.get\_pk(date.today())).sum() |

## Atomics

|  |  |
| --- | --- |
| Django assumes the default DB  everywhere. If you’re writing elsewhere, stick with sane defaults and patterns | **def** atomic\_for\_model(model, \*\*kwargs):  *"""Transactions for the db on a specific model*  *Usage::*  *with atomic\_for\_model(MyModel):*  *...*  *"""*  model\_db = model.objects.all().db  **return** atomic(using=model\_db, \*\*kwargs) |

# Memoization



## Memoization with Decorators

|  |  |
| --- | --- |
| **def** memoize(f):  memo = {}  **def** helper(x):  **if** x **not** **in** memo:  memo[x] = f(x)  **return** memo[x]  **return** helper  **def** fib(n):  **if** n == 0:  **return** 0  **elif** n == 1:  **return** 1  **else**:  **return** fib(n-1) + fib(n-2)  fib = memoize(fib)  **print**(fib(40))  Pythonic  **def** memoize(f):  memo = {}  **def** helper(x):  **if** x **not** **in** memo:  memo[x] = f(x)  **return** memo[x]  **return** helper  @memoize  **def** fib(n):  **if** n == 0:  **return** 0  **elif** n == 1:  **return** 1  **else**:  **return** fib(n-1) + fib(n-2)  **print**(fib(40)) |  |

## Lazy Properties

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a social media post

Description automatically generated

## Global Cache

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

## Instance Cache

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

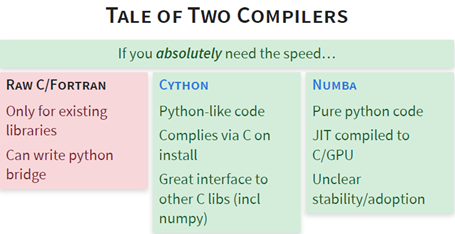
A screenshot of a cell phone

Description automatically generated

# Optimization

A screenshot of a cell phone

Description automatically generated



## Memory Views

A screenshot of a cell phone

Description automatically generated

**C-contiguous**: rows next to each other ⇨ Faster row operations

**F-contiguous** (Fortran): columns next to each other ⇨ Faster column operations

A screenshot of a social media post

Description automatically generated

A screenshot of a cell phone

Description automatically generated

## GIL and Parallelism

A screenshot of a cell phone

Description automatically generated

# Vizualization

A close up of text on a white background

Description automatically generated

## Grammar Graphics

A grammar of graphics is a tool that enables us to concisely describe the components of a graphic. Such a grammar allows us to move beyond named graphics (e.g., the “scatterplot”) and gain insight into the deep structure that underlies statistical graphics — Wickham, A Layered Grammar of Graphics

A screenshot of a social media post

Description automatically generated

## Color and Data Science

A screenshot of a cell phone

Description automatically generated

### Colorspace

|  |  |
| --- | --- |
| Additive color is represented with triplet (R,G,B) in terms of pixels | … which can be mapped into perceptually-uniform Euclidean space e.g., Lab\* |

### Qualitative Color - ColorBrewer

A close up of a map

Description automatically generated

### Sequential Color

You must represent linear data change with linear perceptual change

### Perceptually Uniform Maps

A screenshot of a cell phone

Description automatically generated

A close up of a logo

Description automatically generated

## Divergent

A screenshot of a cell phone

Description automatically generated

## Vizualization as Map Reduce

A screenshot of a cell phone

Description automatically generated

# DECORATOR

A decorator intercepts a function or class after it’s defined. It can wrap the function, alterit, register it or remove it.

## Function as Decorator

### Function as Decorator (w/o arguments)

# PythonDecorators/entry\_exit\_function.py

**def** **entry\_exit**(f):

**def** **new\_f**():

**print**("Entering", f.\_\_name\_\_)

f()

**print**("Exited", f.\_\_name\_\_)

**return** new\_f

**@entry\_exit**

**def** **func1**():

**print**("inside func1()")

**@entry\_exit**

**def** **func2**():

**print**("inside func2()")

func1()

func2()

**print**(func1.\_\_name\_\_)

---------------

Entering func1

inside func1()

Exited func1

---------------

Entering func2

inside func2()

Exited func2

---------------

new\_f

### Function as Decorator (w/ arguments)

# Python Decorators/ decorator\_function\_with\_arguments.py

**def** **decorator\_function\_with\_arguments**(arg1, arg2, arg3):

**def** **wrap**(f):

**print**("1 - Inside wrap()")

**def** **wrapped\_f**(\*args):

**print**("4 - Inside wrapped\_f()")

**print**("5 - Decorator arguments:", arg1, arg2, arg3)

f(\*args)

**print**("7 - After f(\*args)")

**return** wrapped\_f

**return** wrap

**@decorator\_function\_with\_arguments**("hello", "world", **42**)

**def** **sayHello**(a1, a2, a3, a4):

**print**('6 - sayHello arguments:', a1, a2, a3, a4)

**print**("2 - After decoration")

**print**("3 - Preparing to call sayHello()")

sayHello("say", "hello", "argument", "list")

**print**("8 - after first sayHello() call")

sayHello("a", "different", "set of", "arguments")

**print**("9 - after second sayHello() call")

-----------------------------------------

1 - Inside wrap()

2 - After decoration

3 - Preparing to call sayHello()

-----------------------------------------

4 - Inside wrapped\_f()

5 - Decorator arguments: hello world **42**

6 - sayHello arguments: say hello argument list

7 - After f(\*args)

8 - after first sayHello() call

-----------------------------------------

4 - Inside wrapped\_f()

5 - Decorator arguments: hello world **42**

6 - sayHello arguments: a different set of arguments

7 - After f(\*args)

9 - after second sayHello() call

## Class as Decorator

### Class as Decorator (w/o arguments)

With no decorator arguments: to-be-decorated function passed to constructor

# PythonDecorators/my\_decorator.py

**class** **my\_decorator**(object):

**def** **\_\_init\_\_**(self, f):

**print**("1 - inside my\_decorator.\_\_init\_\_()")

f() # Prove that function definition has completed

**def** **\_\_call\_\_**(self):

**print**("4 - inside my\_decorator.\_\_call\_\_()")

**@my\_decorator**

**def** **myFunction**():

**print**("2 - inside myFunction()")

**print**("3 - Finished decorating aFunction()")

myFunction()

**1** - inside my\_decorator.\_\_init\_\_()

**2** - inside myFunction()

**3** - Finished decorating aFunction()

**4** - inside my\_decorator.\_\_call\_\_()

### Class as Decorator (w/ arguments)

With decorator arguments: to-be-decorated function not passed to constructor

**class** **decorator\_with\_arguments**(object):

**def** **\_\_init\_\_**(self, arg1, arg2, arg3):

**print**("1 - Inside \_\_init\_\_()")

self.arg1 = arg1

self.arg2 = arg2

self.arg3 = arg3

**def** **\_\_call\_\_**(self, f):

"""

With decorator arguments, \_\_call\_\_() is only alled once,

as part of the decoration process! You can only give it

a single argument, which is the function object.

"""

**print**("2 - Inside \_\_call\_\_()")

**def** **wrapped\_f**(\*args):

**print**("5 - Inside wrapped\_f()")

**print**("6 - Decorator arguments:", self.arg1, \

self.arg2, self.arg3)

f(\*args)

**print**("8 - After f(\*args)")

**return** wrapped\_f

**@decorator\_with\_arguments**("hello", "world", **42**)

**def** **sayHello**(a1, a2, a3, a4):

**print**('7 - sayHello arguments:', a1, a2, a3, a4)

**print**("3 - After decoration")

**print**("4 - Preparing to call sayHello()")

-----------

sayHello("say", "hello", "argument", "list")

**print**("9 - after first sayHello() call")

-----------

sayHello("a", "different", "set of", "arguments")

**print**("10 - after second sayHello() call")

------------------

1 - Inside \_\_init\_\_()

2 - Inside \_\_call\_\_()

3 - After decoration

4 - Preparing to call sayHello()

------------

5 - Inside wrapped\_f()

6 - Decorator arguments: hello world **42**

7 - sayHello arguments: say hello argument list

8 - After f(\*args)

9 - after first sayHello() call

------------

5 - Inside wrapped\_f()

6 - Decorator arguments: hello world **42**

7 - sayHello arguments: a different set of arguments

8 - After f(\*args)

10 - after second sayHello() call

## REGISTRIES

No need to modify a function, register it!

SUMMARIES = {} # This is a dictionary

**def** **register**(func, name=None):

SUMMARIES[name **or** func.\_\_name\_\_] = func

**return** func

**@register**

**def** **mean**(x):

...

**def** **summarize**(vec, stat='mean'):

**return** SUMMARIES[stat](vec)

### Registering Plugins

This decorator simply registers that a function exists and return it unwrapped. Can be used to create a light-weight plug-in architecture

**import** **random**

PLUGINS = dict()

**def** **register**(func):

"""Register a function as a plug-in"""

PLUGINS[func.\_\_name\_\_] = func

**return** func

**@register**

**def** **say\_hello**(name):

**return** f"Hello {name}"

**@register**

**def** **be\_awesome**(name):

**return** f"Yo {name}, together we are the awesomest!"

**def** **randomly\_greet**(name):

greeter, greeter\_func = random.choice(\

list(PLUGINS.items()))

**print**(f"Using {greeter!r}")

**return** greeter\_func(name)

The @register decorator simply stores a reference to the decorated function in the global PLUGINS dict. Note that you do not have to write an inner function or use @functools.wraps in this example because you are returning the original function unmodified. The randomly\_greet() function randomly chooses one of the registered functions to use. Note that the PLUGINS dictionary already contains references to each function object that is registered as a plugin:

>>> PLUGINS

{'say\_hello': <function say\_hello at **0x7f768eae6730**>,

'be\_awesome': <function be\_awesome at **0x7f768eae67b8**>}

>>> randomly\_greet("Alice")

Using 'say\_hello'

'Hello Alice'

## Using @wraps with decorator

By importing **wraps** from the **functools** module and use it as a decorator for the nested wrapper function inside of **my\_func\_b**,we keep the right name and docstring. In Python interpreter, the help function will work correctly too.

**from** **functools** **import** wraps

**def** **without\_wraps**(func):

**def** **\_\_wrapper**(\*args, \*\*kwargs):

**return** func(\*args, \*\*kwargs)

**return** \_\_wrapper

**def** **with\_wraps**(func):

**@wraps**(func)

**def** **\_\_wrapper**(\*args, \*\*kwargs):

**return** func(\*args, \*\*kwargs)

**return** \_\_wrapper

**@without\_wraps**

**def** **my\_func\_a**():

"""Here is my\_func\_a doc string text."""

**pass**

**@with\_wraps**

**def** **my\_func\_b**():

"""Here is my\_func\_b doc string text."""

**pass**

# Below are the results without using @wraps decorator

**print** my\_func\_a.\_\_doc\_\_

>>> None

**print** my\_func\_a.\_\_name\_\_

>>> \_\_wrapper

# Below are the results with using @wraps decorator

**print** my\_func\_b.\_\_doc\_\_

>>> Here **is** my\_func\_b doc string text.

**print** my\_func\_b.\_\_name\_\_

>>> my\_func\_b

### Functional decorator

Used as @cache(seconds=60), or just @cache

**from** **functools** **import** wraps

**import** **random**

**from** **django.core.cache** **import** cache **as** \_cache

**def** **cache**(\*args, \*\*kwargs):

func = None

**if** len(args) == **1** **and** \_\_builtins\_\_.callable(args[**0**]):

func = args[**0**]

**if** func:

seconds = **60** # default values

**if** **not** func:

seconds = kwargs.get('seconds')

**def** **callable**(func):

**@wraps**(func)

**def** **wrapped**(\*args, \*\*kwargs):

cache\_key = [func, args, kwargs]

result = \_cache.get(cache\_key)

**if** result:

**return** result

result = func(\*args, \*\*kwargs)

\_cache.set(cache\_key, result, timeout=seconds)

**return** result

**return** wrapped

**return** callable(func) **if** func **else** callable

**@cache**(seconds=**60**)

**def** **function\_to\_wrap**(bits=**128**):

**return** random.getrandbits(bits)

**@cache**

**def** **function\_to\_wrap2**(bits=**128**):

**return** random.getrandbits(bits)

**if** \_\_name\_\_ == "\_\_main\_\_":

**print** function\_to\_wrap() # prints '47141457794590517513826129394479136255'

**print** function\_to\_wrap() # prints '47141457794590517513826129394479136255' also (cached)

**print** function\_to\_wrap2(**32**) # prints '2202905596'

**print** function\_to\_wrap2(**32**) # prints '2202905596' also (cached)

### Decorator In class style

Decorator can be a callable or return a callable.

**from** **functools** **import** wraps

**import** **random**

**from** **django.core.cache** **import** cache **as** \_cache

**class** **cache**(object):

**def** **\_\_init\_\_**(self, seconds=None):

self.seconds = seconds

**def** **\_\_call\_\_**(self, func):

**@wraps**(func)

**def** **callable**(\*args, \*\*kwargs):

cache\_key = [func, args, kwargs]

result = \_cache.get(cache\_key)

**if** result:

**return** result

result = func(\*args, \*\*kwargs)

\_cache.set(cache\_key, result, timeout=self.seconds)

**return** result

**return** callable

**@cache**(seconds=**60**)

**def** **function\_to\_wrap**(bits=**128**):

**return** random.getrandbits(bits)

**if** \_\_name\_\_ == "\_\_main\_\_":

**print** function\_to\_wrap() # prints '47141457794590517513826129394479136255'

**print** function\_to\_wrap() # prints '47141457794590517513826129394479136255' also (cached)

# CONTEXT MANAGER

Context managers let you allocate and release resources precisely when you want to. Most widely used with statement.

**with** open('some\_file', 'w') **as** opened\_file:

opened\_file.write('Hola!')

is equivalent to:

file = open('some\_file', 'w')

**try**:

file.write('Hola!')

**finally**:

file.close()

## Context Manager as a Class

A context manager has an \_\_enter\_\_ and \_\_exit\_\_ method.

**class** **File**(object):

**def** **\_\_init\_\_**(self, file\_name, method):

self.file\_obj = open(file\_name, method)

**def** **\_\_enter\_\_**(self):

**return** self.file\_obj

**def** **\_\_exit\_\_**(self, type, value, traceback):

self.file\_obj.close()

With \_\_enter\_\_ and \_\_exit\_\_, class File can be used in a with statement.

**with** File('demo.txt', 'w') **as** opened\_file:

opened\_file.write('Hola!')

Context Manager class requires \_\_exit\_\_ method to accept 3 arguments.

1. The with statement stores the \_\_exit\_\_ method of the File class.
2. It calls the \_\_enter\_\_ method of the File class.
3. The \_\_enter\_\_ method opens the file and returns it.
4. The opened file handle is passed to opened\_file.
5. We write to the file using .write().
6. The with statement calls the stored \_\_exit\_\_ method.
7. The \_\_exit\_\_ method closes the file.

## Handling Exceptions

Between the 4th and 6th step, if an exception occurs, Python passes the type, value and traceback of the exception to the \_\_exit\_\_ method. It allows the \_\_exit\_\_ method to decide how to close the file and if any further steps are required. Steps taken by the with statement when an error is encountered:

1. It passes the type, value and traceback of the error to the \_\_exit\_\_ method.
2. It allows the \_\_exit\_\_ method to handle the exception.
3. If \_\_exit\_\_ returns True then the exception was gracefully handled.
4. If anything other than True is returned by the \_\_exit\_\_ method then the exception is raised by the with statement.

\_\_exit\_\_ returns None (when no return statement is encountered then the method returns None). Therefore, the with statement raises the exception:

Traceback (most recent call last):

File "<stdin>", line **2**, **in** <module>

**AttributeError**: 'file' object has no attribute 'undefined\_function'

Let’s try handling the exception in the \_\_exit\_\_ method:

**class** **File**(object):

**def** **\_\_init\_\_**(self, file\_name, method):

self.file\_obj = open(file\_name, method)

**def** **\_\_enter\_\_**(self):

**return** self.file\_obj

**def** **\_\_exit\_\_**(self, type, value, traceback):

**print**("Exception has been handled")

self.file\_obj.close()

**return** True

**with** File('demo.txt', 'w') **as** opened\_file:

opened\_file.undefined\_function()

# Output: Exception has been handled

Our \_\_exit\_\_ returned True: no exception was raised by the with statement.

## Context Manager as Generator

To implement Context Managers using decorators and generators, use **contextlib module**. Instead of class, implement Context Manager using a generator function.

**from** **contextlib** **import** contextmanager

**@contextmanager**

**def** **open\_file**(name):

f = open(name, 'w')

**yield** f

f.close()

1. Python encounters the yield keyword. Due to this it creates a generator instead of a normal function.
2. Due to the decoration, contextmanager is called with the function name (open\_file) as it’s argument.
3. The contextmanager decorator returns the generator wrapped by the GeneratorContextManager object.
4. The GeneratorContextManager is assigned to the open\_file function. Therefore, when we later call the open\_file function, we are actually calling the GeneratorContextManager object.

### General Form

**from** **contextlib** **import** contextmanager

**@contextmanager**

**def** **some\_context**(\*a, \*\*k):

context = something(\*a, \*\*k)

**try**:

**yield** context

**except** Exception\_1:

handler\_1

**except** Exception\_n:

handler\_n

**finally**:

No matter what happened previously, the *final-block* is executed once the code block is complete and any raised exceptions handled. Even if there's an error in an exception handler or the *else-block* and a new exception is raised, the code in the *final-block* is still run

### Sequence

**from** **contextlib** **import** contextmanager

**@contextmanager**

**def** **write\_signature**(\*args, \*\*kwargs):

**with** open(\*args, \*\*kwargs) **as** f:

**yield** f

f.write('**\n\n**Sincerely,**\n\n**-Scott')

**with** write\_signature('letter.txt', mode='w') **as** f:

f.write("Hello!")

Hello!

Sincerely,

-Scott

# ITERATOR

## Array

>>> a

array([ **0.** , **0.5**, **1.** ])

>>> **2**\*\*a

array([ **1.**, **1.41421356**, **2.**])

>>> np.array([[**0**, **1**]]) \ # dim (1, 2)

+ np.array([[**0**], [**1**]]) # dim (2, 1)

array([[**0**, **1**], # dim (2, 2)

[**1**, **2**]])

## Generator

**def** **iter\_range**(n):

i = **0**

**while** i < n:

**yield** i

i += **1**

# Initialize the list

my\_list = [**1**, **3**, **6**, **10**]

# square each term using **list comprehension**

# Output: [1, 9, 36, 100]

**[**x\*\***2** **for** x **in** my\_list**]**

# Similar results with **generator expression**

# Output: <generator object <genexpr> at 0x0000000002EBDAF8>

**(**x\*\***2** **for** x **in** my\_list**)**

## Map

**map(f, iter) === (f(\_) for \_ in iter)**

# Return double of n

**def** **addition**(n):

**return** n + n

# We double all numbers using map()

numbers = (**1**, **2**, **3**, **4**)

result = map(addition, numbers)

# Double all numbers using map and lambda

numbers = (**1**, **2**, **3**, **4**)

result = map(**lambda** x: x + x, numbers)

Output :

{**2**, **4**, **6**, **8**}

# Add two lists using map and lambda

numbers1 = [**1**, **2**, **3**]

numbers2 = [**4**, **5**, **6**]

result = map(**lambda** x, y: x + y, numbers1, numbers2)

Output :

[**5**, **7**, **9**]

# List of strings

l = ['sat', 'bat', 'cat', 'mat']

# map() can list the strings individually

test = list(map(list, l))

Output :

[['s', 'a', 't'], ['b', 'a', 't'], ['c', 'a', 't']

## Reduce

from functools import reduce

reduce(**lambda** x,y: x+y, [**47**,**11**,**42**,**13**])

**113**

f = **lambda** a,b: a **if** (a > b) **else** b

reduce(f, [**47**,**11**,**42**,**102**,**13**])

**102**

reduce(**lambda** x, y: x+y, range(**1**,**101**))

**5050**

## Zip

numbersList = [**1**, **2**, **3**]

strList = ['one', 'two']

numbersTuple = ('ONE', 'TWO', 'THREE', 'FOUR')

result = zip(numbersList, numbersTuple)

# Converting to set

resultSet = set(result)

**print**(resultSet)

result = zip(numbersList, strList, numbersTuple)

# Converting to set

resultSet = set(result)

**print**(resultSet)

{(**2**, 'TWO'), (**3**, 'THREE'), (**1**, 'ONE')}

{(**2**, 'two', 'TWO'), (**1**, 'one', 'ONE')}

## Filter

**filter(f, iter) === (\_ for \_ in iter if f(\_))**

# function that filters vowels

**def** **fun**(variable):

letters = ['a', 'e', 'i', 'o', 'u']

**if** (variable **in** letters):

**return** True

**else**:

**return** False

# sequence

sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']

# using filter function

filtered = filter(fun, sequence)

**print**('The filtered letters are:')

**for** s **in** filtered:

**print**(s)

The filtered letters are:

e

e

Normally used **with** Lambda functions to separate list, tuple, sets

# List contains both even and odd numbers.

seq = [**0**, **1**, **2**, **3**, **5**, **8**, **13**]

# result contains odd numbers of the list

result = filter(**lambda** x: x % **2**, seq)

**print**(list(result))

# result contains even numbers of the list

result = filter(**lambda** x: x % **2** == **0**, seq)

**print**(list(result))

# CLASS

## Unpacking (\*args and \*\*kwargs)

**def** f (x, y, z) :

**return** [x, y, z]

t = (3, 4) *# Tuple aka a sequence of immutable Python objects*

### Unpacking argument lists

**assert** f(2, t, 5) == [2, (3, 4), 5]

**assert** f(2, 5, t) == [2, 5, (3, 4)]

**assert** f(2, \*t) == [2, 3, 4]

**assert** f(z = 2, \*t) == [3, 4, 2] *# Values for X, Y passed through tuple*

**assert** f(\*t, z = 2) == [3, 4, 2] *# Values for X, Y passed through tuple*

f(\*t) *# TypeError: f() missing 1 required positional argument: 'z'*

f(2, 3, \*t) *# TypeError: f() takes 3 positional arguments but 4 given*

f(\*t, 2) *# SyntaxError: only named arguments may follow \*expression*

f(x = 2, \*t) *# TypeError: f() got multiple values for argument 'x'*

### kwarg (keyword argument)

d = {"z" : 4, "y" : 3, "x" : 2}

**assert** f(\*\*d) == [2, 3, 4]

f(2, \*\*d) *# TypeError: f() got multiple values for argument 'x'*

f(x = 2, \*\*d) *# TypeError: f() got multiple values f/ keyword argument 'x'*

d = {"z" : 4, "y" : 3}

**assert** f(2, \*\*d) == [2, 3, 4]

*# f(\*\*d, 2) # SyntaxError: invalid syntax*

**assert** f(x = 2, \*\*d) == [2, 3, 4]

**assert** f(\*\*d, x = 2) == [2, 3, 4]

d = {"y" : 3}

**assert** f(2, z = 4, \*\*d) == [2, 3, 4]

**assert** f(2, \*\*d, z = 4) == [2, 3, 4]

### args and \*kwargs together

t = (3,)

d = {"z" : 4}

**assert** f(2, \*t, \*\*d) == [2, 3, 4]

**assert** f(y = 3, \*t, \*\*d) == [3, 3, 4]

**assert** f(\*t, y = 3, \*\*d) == [3, 3, 4]

**assert** f(\*t, \*\*d, y = 3) == [3, 3, 4]

*# Bad Example: argument lists try to fill arguments out in order.*

*# TypeError: f() got multiple values for argument 'x'*

**assert** f(x = 2, \*t, \*\*d) == [2, 3, 4]

## General

*""" Use 'mysillyobject', 'abc' instead of self"""*

**class** Person:

**def** \_\_init\_\_(mysillyobject, name, age):

mysillyobject.name = name

mysillyobject.age = age

**def** myfunc(abc):

**print**("Hello my name is " + abc.name)

p1 = Person("John", 36)

p1.myfunc()

**class** Animal:

**def** \_\_init\_\_(self, \*\*kwargs):

self.species = kwargs.get("species")

self.age = kwargs.get("age")

self.sound = kwargs.get("sound")

>>> wolf = Animal(species="Canus L", age=5, sound="howl", color="gray")

>>> wolf.species

"Canus L"

## Inherit and Override

**class** **Regressor**:

**def** **\_\_init\_\_**(self, penalty):

self.penalty = penalty

**def** **predict**(self, x):

**return** np.dot(x, self.beta)

**class** **Classifier**(Regressor):

**def** **predict**(self, x):

# Go up the chain

**return** super().predict(x) >= **0**

## Abstracts and Interfaces

### Informal

**class** **BaseRegressor**:

**def** **regress**(self, x):

**raise** **NotImplementedError**('Abstract')

>>> glm = GLM()

>>> isinstance(glm, BaseSGDRegressor)

True

>>> glm.regress(x)

### Formal: use ABC

**from** **abc** **import** ABC, abstractmethod

**class** **BaseRegressor**(ABC):

**@abstractmethod**

**def** **regress**(self):

**raise** **NotImplementedError**('Abstract')

>>> BaseRegressor()

**TypeError**: Cannot instantiate abstract **class**

**BaseRegressor** **with** abstract methods regress

## Mixins

Capture orthogonal properties + methods: mix them in! **Mixin** = class with no data, only methods, (normally) no \_\_init\_\_()🢥 class inheriting mixin does not need to use super()

**from** **django.http** **import** JsonResponse

**class** **JSONResponseMixin**(object):

""" Mixin to render a JSON response"""

**def** **render\_to\_json\_response**(self, context, \*\*response\_kwargs):

""" Returns JSON response transform 'context' to make payload """

**return** JsonResponse(self.get\_data(context), \*\*response\_kwargs)

**def** **get\_data**(self, context):

""" Returns object serialized as JSON by json.dumps() """

**return** context

Now, pass it in to a class, call methods: self.render\_to\_response(...).

**from** **django.views.generic** **import** TemplateView

**class** **JSONView**(JSONResponseMixin, TemplateView):

**def** **render\_to\_response**(self, context, \*\*response\_kwargs):

**return** self.render\_to\_json\_response(context, \*\*response\_kwargs)

## Composition

Composition means that an object knows another object, and explicitly delegates some tasks to it. While inheritance is implicit, composition is explicit. Example of classic composition, making an object part of the other as an attribute

**class** SecurityDoor:

colour = 'gray'

locked = True

**def** \_\_init\_\_(self, number, status):

self.door = Door(number, status)

**def** open(self):

**if** self.locked:

**return**

self.door.open()

**def** close(self):

self.door.close()

SecurityDoor is an object and no more a Door 🢥 Internal structure of Door not copied. The composed SecurityDoor must redefine colour attribute. Delegation applies to methods. How about attributes? Python provides indirection for objects manipulation, including attribute access. Accessing attributes is ruled by \_\_getattribute\_\_()- called whenever an object attribute is accessed. Overriding \_\_getattribute\_\_()is overkill; it is a very complex method, and, being called on every attribute access, any change makes the whole thing slower. The method to leverage is \_\_getattr\_\_()- special method called whenever requested attribute not found in object. So basically it is the right place to dispatch all attribute and method access our object cannot handle. The previous example becomes

**class** SecurityDoor:

locked = True

**def** \_\_init\_\_(self, number, status):

self.door = Door(number, status)

**def** open(self):

**if** self.locked:

**return**

self.door.open()

**def** \_\_getattr\_\_(self, attr):

**return** getattr(self.door, attr)

Using \_\_getattr\_\_() blends the separation line between inheritance and composition since after all the former is a form of automatic delegation of every member access.

**class** ComposedDoor:

**def** \_\_init\_\_(self, number, status):

self.door = Door(number, status)

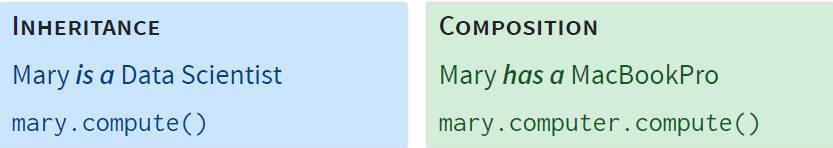
**def** \_\_getattr\_\_(self, attr):

**return** getattr(self.door, attr)

**Note**: getattr() which is different from \_\_getattr\_\_(). The former is a built-in that is equivalent to the dotted syntax, i.e. getattr(obj, 'someattr') is the same as obj.someattr, but you have to use it since the name of the attribute is contained in a string. Composition provides a superior way to manage delegation since it can selectively delegate the access, even mask some attributes or methods, while inheritance cannot. In Python you also avoid the memory problems that might arise when you put many objects inside another; Python handles everything through its reference, i.e. through a pointer to the memory position of the thing, so the size of an attribute is constant and very limited.

## Composition vs. Inheritance

Composition gives classes rich properties, and delegate responsibilities to them



### Composition as Instance Properties

**class** **Student**:

**def** **\_\_init\_\_**(self, computer=None):

self.computer = computer **or** Laptop()

**def** **compute**(self, \*args):

**return** self.computer.compute(\*args)

### Composition as Declarative Classes

**class** **Student**:

COMPUTER\_CLASS = Laptop

**def** **\_\_init\_\_**(self):

self.computer = self.COMPUTER\_CLASS()

**class** **MacStudent**(Student):

COMPUTER\_CLASS = MacBookPro

**import** **time**

**from** **functools** **import** wraps

**def** **rv\_decorator**(func):

**@wraps**(func)

**def** **wrapper**(\*args, \*\*kwds):

rv = func(\*args, \*\*kwds)

print(func.\_\_name\_\_)

print(rv)

**return** rv

**return** wrapper

**class** **Computer**:

# Computer is a general purpose computing device

**def** **\_\_init\_\_**(self, computer\_type='', year=**1995**):

self.computer\_type = computer\_type

self.speed = 'Fast'

**@rv\_decorator**

**def** **add\_ints**(self, a, b):

**return** a+b

**@rv\_decorator**

**def** **divide\_ints**(self, a, b):

**return** a//b

**class** **Calculator**(Computer):

# Calculator \*is a\* specific type of Computer

# set calculator speed

speed = 'Slow'

**@rv\_decorator**

**def** **add\_ints**(self, a, b):

time.sleep(**3**)

**pass**

calc = Calculator('TI-84', **2013**)

SmartWatch implements Calculator add\_ints through composition

**class** **SmartWatch**():

**def** **\_\_init\_\_**(self, computer\_type='', year=**2015**):

self.\_cpu = Computer(computer\_type,year)

**def** **add\_ints**(self, a, b):

self.\_cpu.add\_ints(a,b)

watch = SmartWatch('Apple', **2019**)

## Declarative Classes

Capture logic through composition, declare values and config on class. Minimize unique functions and complex inheritance

**class** **SomeClass**:

a = A()

b = B()

**class** **Variant**(SomeClass):

b = B(val=**2**)

[**attr.s**](https://github.com/python-attrs/attrs)

**import** **attr**

**@attr.s**

**class** **SomeClass**(object):

a\_number = attr.ib(default=**42**)

list\_of\_numbers = attr.ib(factory=list)

...

**def** **hard\_math**(self, another\_number):

**return** self.a\_number + sum(self.list\_of\_numbers) \* another\_number

sc = SomeClass(**1**, [**1**, **2**, **3**])

sc

SomeClass(a\_number=**1**, list\_of\_numbers=[**1**, **2**, **3**])

sc.hard\_math(**3**)

**19**

sc == SomeClass(**1**, [**1**, **2**, **3**])

True

sc != SomeClass(**2**, [**3**, **2**, **1**])

True

attr.asdict(sc)

{'a\_number': **1**, 'list\_of\_numbers': [**1**, **2**, **3**]}

SomeClass()

SomeClass(a\_number=**42**, list\_of\_numbers=[])

## Instance, Class, Static Methods

**class** **MyClass**:

**def** **method**(self):

**return** 'instance method called', self

**@classmethod**

**def** **classmethod**(cls):

**return** 'class method called', cls

**@staticmethod**

**def** **staticmethod**():

**return** 'static method called'

Instead of parameter self, **class methods** take parameter cls (points to class, not object instance). **Static method** takes neither (cannot modify object or class)

>>> MyClass.classmethod()

('class method called', <**class** **MyClass** at **0x101a2f4c8**>)

>>> MyClass.staticmethod()

'static method called'

>>> MyClass.method()

**TypeError**: unbound method method() must

be called **with** MyClass instance **as** first

argument (got nothing instead)

### Class Method example

**class** **Pizza**:

**def** **\_\_init\_\_**(self, ingredients):

self.ingredients = ingredients

**def** **\_\_repr\_\_**(self):

**return** f'Pizza({self.ingredients!r})'

>>> Pizza(['cheese', 'tomatoes'])

Pizza(['cheese', 'tomatoes'])

## Properties

### @property

|  |  |
| --- | --- |
| ***Encapsulation …***  **class** **A**:  **def** **\_\_init\_\_**(self, x):  self.\_x = x  **def** **get\_x**(self): # Post-process x  **return** self.\_x  **def** **set\_x**(self, val): # Pre-process x  self.\_x = val | ***… Can be hidden***  **class** **A**:  **@property**  **def** **x**(self):  **return** self.\_x  **@x.setter**  **def** **x**(self, val):  self.\_x = val  >>> A(**5**).x # not .x() |

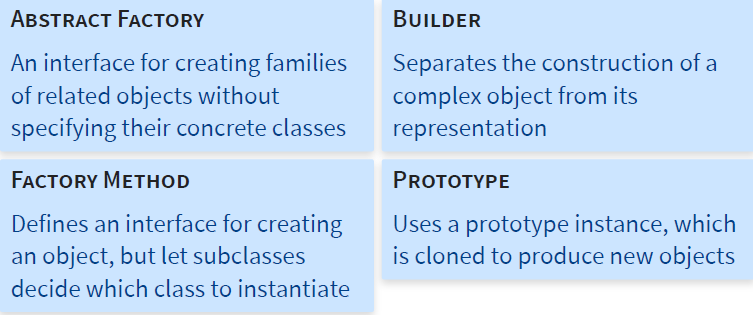
### @property is descriptor

|  |  |  |  |
| --- | --- | --- | --- |
| ***Property …***  **class** **A**:  **@property**  **def** **x**(self):  # access as a.x, not a.x()  **return** self.\_x  >>> A.x  <property at **0x10dd1f9a8**>  >>> A(**5**).x  **5** | ***… is shorthand for***  **class** **X**:  **def** **\_\_get\_\_**(self, obj, objtype):  **if** obj **is** None: # Invoked via `A.x`  **return** self  **return** obj.\_x # Invoked via `A().x`  **class** **A**:  x = X()  >>> A.x # == x.\_\_get\_\_(None, A)  <\_\_main\_\_.X at **0x10dcccb38**>  >>> A(**5**).x #x.\_\_get\_\_(A(5), A)  **5** | | |
| **def** **logged\_property**(method):  **@property**  **@wraps**(method)  **def** **wrapped**(self):  logger.info(  'Accessing {}'.format(method.\_\_name\_\_))  **return** method(self)  **return** wrapped | | **class** **A**:  **@logged\_property**  **def** **x**(self):  **return** self.\_x  >>> A(**5**).x  Accessing x  **5** | |
| **class** **RevealAccess**():  """Logs during access"""  **def** **\_\_set\_name\_\_**(self, owner, name):  self.name = name  **@property**  **def** **attr**(self):  **return** f'\_{self.name}'  **def** **\_\_get\_\_**(self, obj, objtype):  **if** obj **is** None:  **return** self  logger.info(f'Retrieving {self.name}')  **return** getattr(obj, self.attr)  **def** **\_\_set\_\_**(self,obj,val):  logger.info(f'Updating {self.name}')  setattr(obj, self.attr, val) | | | **class** **MyClass**(object):  x = RevealAccess()  >>> m = MyClass()  >>> m.x = **20**  Updating x  >>> m.x  Retrieving var x  **20**  *Note that \_\_set\_name\_\_ allows x to know the name it is assigned on MyClass!* |

## CLASS as Constructors

|  |  |
| --- | --- |
|  |  |

# Creation Design Patterns



## Simple Factories

|  |  |
| --- | --- |
| Functions designed to construct data blobs for testing classifiers. “Simple” factories are just functions that do a good job of generalizing how you might want to construct a piece of data.  They’re not really “advanced python” but they are a good design principle | *# sklearn.datasets*  **def** make\_classification(n, p, nclasses, ...):  *"""Generate random classification*  *:returns: x[n, p], y[n]*  *"""*  **def** test\_clf():  x, y = make\_classification(100, 2)  clf = MyClassifier()  clf.train(x, y) |

## Prototype Factories

|  |  |
| --- | --- |
| **Eg luigi’s Task.clone**  A “prototype” pattern creates new object similar to a  reference, or prototype, object. Some differences it injects, like a different class, modified params,etc. | **class** Task:  **def** clone(self, other\_cls, \*\*kwargs):  new\_k = {}  **for** param\_name, param\_class **in** other\_cls.get\_params():  **if** param\_name **in** kwargs:  new\_k[param\_name] = kwargs[param\_name]  **elif** hasattr(self, param\_name):  new\_k[param\_name] = getattr(self, param\_name)  **return** other\_cls(\*\*new\_k) |
| **… or our**  **HistoryRequirement** | **class** HistoryRequirement(Requirement):  **def** \_\_call\_\_(self, task):  d = task.date  **return** [  task.clone(self.other\_task, date=d-timedelta(days-i-1))  **for** i **in** range(self.ndays)] |

## Factory Methods

|  |  |
| --- | --- |
| Best achieved with @classmethod. Provide a new, common way to  create instances of any subclass | **class** WordEmbedding:  @classmethod  **def** from\_files(cls, word\_file, vec\_file):  *"""Instantiate an embedding from files"""*  *# note cls vs WordEmbedding*  **return** cls(  load\_vocab(word\_file), np.load(vec\_file)) |

## Builders

|  |  |
| --- | --- |
| … are classes designed to  iteratively build another object | **class** Car:  ...  **class** CarBuilder(Builder):  **def** \_\_init\_\_(self):  self.car = Car()  **def** set\_wheels(self, value):  self.car.wheels = value  **return** self  **def** get\_result(self):  **return** self.car  >>> CarBuilder().set\_wheels('dubs').get\_result()  Car() |
| Not a common DS paradigm  IMHO, except when you consider  internal objects like DAG’s or  QuerySet().query | >>> ddf = dask.dataframe.read\_parquet('data/grad')  >>> ddf.groupby('grad').sum().dask  {  ('dataframe-groupby-sum-agg-9939d0', 0): ...  ...  }  >>> MyModel.objects.filter(name="scott").query  SELECT \* **from** **myapp\_mymodel** **WHERE** **name** = 'scott' |

## Abstract Factories

|  |  |
| --- | --- |
| Class designed to generate other  Classes - Usually subclass or parameterize to define the class to create | **class** DatasetGenerator:  GENERATOR = None *# Abstract*  **def** \_\_init\_\_(self, \*\*make\_kwargs):  self.make\_kwargs = make\_kwargs  **def** \_\_call\_\_(self):  **return** self.GENERATOR(\*\*self.make\_kwargs)  **class** ClassificationGenerator(DatasetGenerator):  GENERATOR = make\_classification  >>> factory = ClassificationGenerator(nclasses=3)  >>> factory()  ([[...]], [...]) |

## Factory Boy

|  |  |
| --- | --- |
| Easy factories for ORM’s (and any  class) - Claims to be for testing, but is useful in general as well! | **from** **factory** **import** Factory  **class** UserFactory(Factory):  **class** Meta:  model = User  firstname = "John"  lastname = "Doe"  >>> john = UserFactory()  <User: John Doe> |
| Note that syntax!  Factory Boy made a class constructor return an instance of  a different class!! | **class** FactoryMetaClass(type):  *# \_\_call\_\_ as a classmethod!*  **def** \_\_call\_\_(cls, \*\*kwargs):  *# Normally, cls() would create an*  *# instance of the class*  **if** cls.\_meta.strategy == BUILD\_STRATEGY:  **return** cls.build(\*\*kwargs)  ...  *# cf (not how they implemented it!):*  >>> factory = UserFactory()  *# instance.\_\_call\_\_()*  >>> john = factory()  <User: John Doe> |
| Arguably, they should have  separated defaults and  constructors  I would have generated a factory  instance, which itself generates  model instances (and allows for  parameterizing the factory!) | *# Not how they did it!*  *# factory definition: schema/model etc*  **class** UserFactory(Factory):  **class** Meta:  model = User  *# Create defaults with an instance, not subclass*  >>> factory = UserFactory(firstname='John')  >>> factory() *# Fake/random data*  <User: John Kennedy>  >>> factory(lastname='Doe') *# Overrides*  <User: John Doe>  >>> factory(lastname='Connor')  <User: John Connor>  ...  *# cf (not how they implemented it!):*  >>> factory = UserFactory()  *# instance.\_\_call\_\_()*  >>> john = factory()  <User: John Doe> |
| Designed for Django, but  promoted/generalized to work  with any ORM or object type | **class** OrderTests(TestCase):  **def** test\_orders(self):  order = OrderFactory(  amount=200,  status='PAID',  *# Automatically creates FK's*  customer\_\_is\_vip=True,  address\_\_country='AU',  )  *# Run the tests here* |



## Faker

|  |  |
| --- | --- |
| Factory Boy uses [Faker](https://faker.readthedocs.io/en/master/) to help stub  out fake data for testing | **from** **faker** **import** Faker  fake = Faker()  >>> fake.name()  'Lucy Cechtelar'  >>> fake.address()  '426 Jordy Lodge Cartwrightshire, SC 88120-6700' |

🞟Classes provide reusable pairings of data and functions. 🞟Factories provide reusable constructors for classes that can encapsulate various and sophisticated creation strategies. 🞟If MyClass() isn’t enough to create a good default instance for you, consider a factory

# Python

* We have repeatable and appropriately specified virtual environments
* We can choose between minimal reqs and frozen dependencies
* We understand the role of pipenv for library and app development

# Testing

* We know the importance of unit testing, how to test, and what to test
* We can measure coverage, including code branches
* We know the environment matters, and test libraries where it counts
* We can mock out code, even faking integration with other systems

# Workflows

* We understand the context of each change - fixes, features, and breaks
* We track history linearly and meaningfully using semantic versions
* We bootstrap our work and codify best practices with templates

# Higher Levels

* We recognize meta-patterns in code at levels higher than a function
* We wrap, register, and alter functions using decorators
* We provide reusable context to code using context managers

# Deployment

* We know what is part of our code, and what is not
* We configure our deployments with environment variables, ensuring our code is useable anywhere
* We handle data and secrets with discretion and privacy

# Looping

* We see past for loops and recognize the higher looping primitive
* We can write stateless, functional code that expresses what we want, without telling the machine what to do
* We know the tradeoffs between efficiency, clarity, and diagnosability
* We know why we iterate, why we map, and why we reduce

# Functional Coding

* We understand that our code is data, and may be operated on
* We know when state is valuable, and when it fails us
* We can encapsulate logic in functions that we pass as arguments to higher frameworks
* We strive to be declarative in all that we do

# Composition

* We look for new ways to simplify and reuse our logic, such as mixins and composition
* We can add rich, reusable, and declarative properties with descriptors

# Graphical Programs

* We recognize our programs, literally, as directed graphs
* We can directly visualize, debug, and optimize a graph
* We can construct graphs to represent our ideas and programs
* We can scaffold our application at the highest level using dataflows
* We can solve new and extremely important problems using the graph, such as seen with Salted