Contents

[Asking for Help 2](#_Toc8111990)

[PYTHON @ HIGHER LEVEL 2](#_Toc8111991)

[REPEATABLE ENVIRONMENT 2](#_Toc8111992)

[CI/CD 2](#_Toc8111993)

[Testing 2](#_Toc8111994)

[STRATEGY for Data Science 2](#_Toc8111995)

[More ON Testing 2](#_Toc8111996)

[PYTEST Fixtures 2](#_Toc8111997)

[PYTEST PLUGINS Coverage 2](#_Toc8111998)

[Test Driven Development 3](#_Toc8111999)

[MOCK 3](#_Toc8112000)

[MOTO 3](#_Toc8112001)

[TEST STRUCTURE 3](#_Toc8112002)

[TEST MATRIX 3](#_Toc8112003)

[CONFIG 3](#_Toc8112004)

[PIPENV 3](#_Toc8112005)

[Environments, Apps, Libraries 3](#_Toc8112006)

[Virtual Environments 4](#_Toc8112007)

[What are your REQS? 4](#_Toc8112008)

[Why PIPENV? 4](#_Toc8112009)

[CONDA 4](#_Toc8112010)

[Libraries versus Applications 4](#_Toc8112011)

[SEMVER 4](#_Toc8112012)

[REQUIREMENTS w.r.t SEMVER 4](#_Toc8112013)

[Tracking the version 4](#_Toc8112014)

[Keep Version Consistent 4](#_Toc8112015)

[SETUPTOOLS\_SCM 4](#_Toc8112016)

[DOCKER 4](#_Toc8112017)

[Build vs. Mounting 5](#_Toc8112018)

[Build vs. Runtime 5](#_Toc8112019)

[LOGGING 5](#_Toc8112020)

[Functional Programming 5](#_Toc8112021)

[STATE 5](#_Toc8112022)

[DECLARATIVE vs. PROCEDURAL 5](#_Toc8112023)

[LUIGI 5](#_Toc8112024)

[Atomicity 5](#_Toc8112025)

[Tasks 5](#_Toc8112026)

[Task Skeleton 5](#_Toc8112027)

[Parameters 6](#_Toc8112028)

[Atomic Targets 6](#_Toc8112029)

[Task Types 6](#_Toc8112030)

[External Programs 6](#_Toc8112031)

[Output 6](#_Toc8112032)

[Requirements 6](#_Toc8112033)

[APACHE PARQUET/ARROW 6](#_Toc8112034)

[Column Stores 7](#_Toc8112035)

[DASK 7](#_Toc8112036)

[Result in Pieces 7](#_Toc8112037)

[Split, Apply, Combine 7](#_Toc8112038)

[Dask.DataFrame 7](#_Toc8112039)

[HASH 7](#_Toc8112040)

[SALTED GRAPHS 7](#_Toc8112041)

[Data Dependency Hell 7](#_Toc8112042)

[Content Addressable Filesystems 7](#_Toc8112043)

[Versioned DAG 8](#_Toc8112044)

[Minimal Implementation 8](#_Toc8112045)

[… With Considerations 8](#_Toc8112046)

[The Signature Input 8](#_Toc8112047)

[Atomic Writes 8](#_Toc8112048)

[DECORATOR 9](#_Toc8112049)

[Function as Decorator 9](#_Toc8112050)

[Function as Decorator (w/o arguments) 9](#_Toc8112051)

[Function as Decorator (w/ arguments) 9](#_Toc8112052)

[Class as Decorator 9](#_Toc8112053)

[Class as Decorator (w/o arguments) 9](#_Toc8112054)

[Class as Decorator (w/ arguments) 9](#_Toc8112055)

[REGISTRIES 9](#_Toc8112056)

[Registering Plugins 9](#_Toc8112057)

[Using @wraps with decorator 10](#_Toc8112058)

[Functional decorator used as @cache(seconds=60), or just @cache 10](#_Toc8112059)

[In class style - decorator can be a callable or return a callable. 10](#_Toc8112060)

[CONTEXT MANAGER 10](#_Toc8112061)

[Context Manager as a Class 10](#_Toc8112062)

[Handling Exceptions 10](#_Toc8112063)

[Context Manager as Generator 10](#_Toc8112064)

[General Form 11](#_Toc8112065)

[Sequence 11](#_Toc8112066)

[ITERATOR 11](#_Toc8112067)

[Array 11](#_Toc8112068)

[Generator 11](#_Toc8112069)

[Map 11](#_Toc8112070)

[Reduce 11](#_Toc8112071)

[Zip 11](#_Toc8112072)

[Filter 11](#_Toc8112073)

[CLASS 11](#_Toc8112074)

[Inherit and Override 11](#_Toc8112075)

[Abstracts and Interfaces 11](#_Toc8112076)

[Informal 11](#_Toc8112077)

[Formal: use ABC 11](#_Toc8112078)

[Mixins 11](#_Toc8112079)

[Composition vs. Inheritance 12](#_Toc8112080)

[Composition as instance properties 12](#_Toc8112081)

[Composition as declarative classes 12](#_Toc8112082)

[Declarative Classes 12](#_Toc8112083)

[Instance, Class and Static Methods 12](#_Toc8112084)

[Class Method example 12](#_Toc8112085)

[Properties 12](#_Toc8112086)

[@property 12](#_Toc8112087)

[@property is descriptor 12](#_Toc8112088)

[Python 14](#_Toc8112089)

[Testing 14](#_Toc8112090)

[Workflows 14](#_Toc8112091)

[Higher Levels 14](#_Toc8112092)

[Deployment 14](#_Toc8112093)

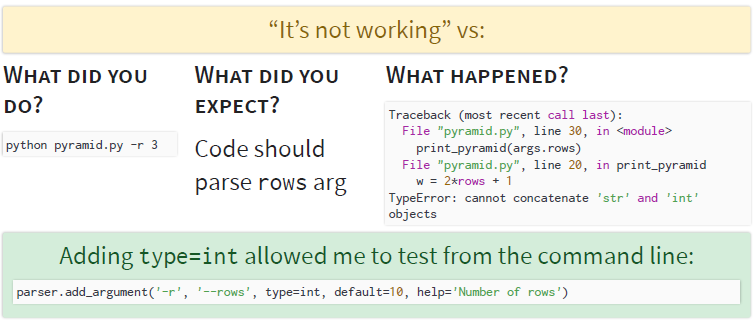
[Looping 14](#_Toc8112094)

[Functional Coding 14](#_Toc8112095)

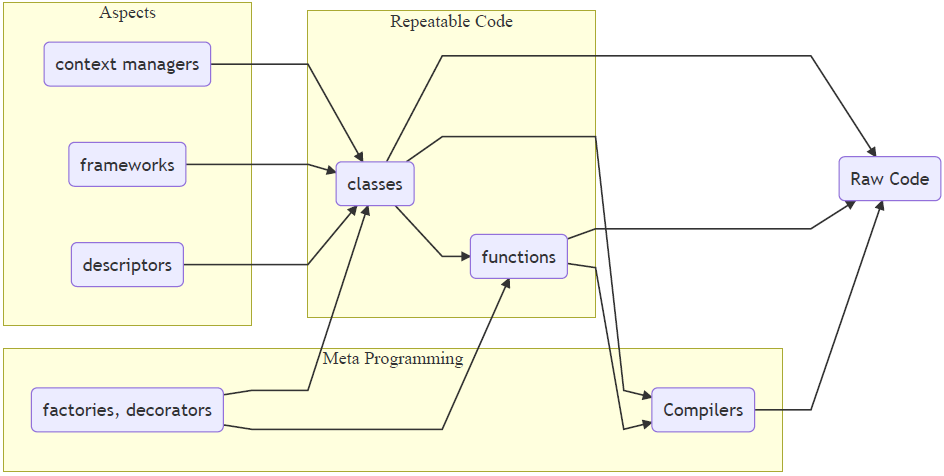
[Composition 14](#_Toc8112096)

[Graphical Programs 14](#_Toc8112097)

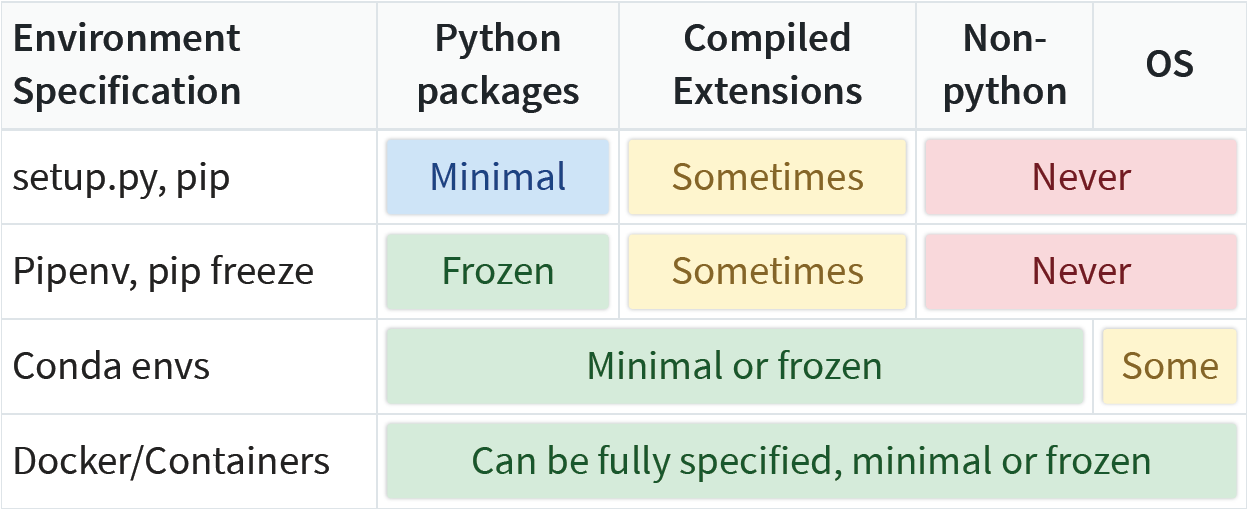
# Asking for Help



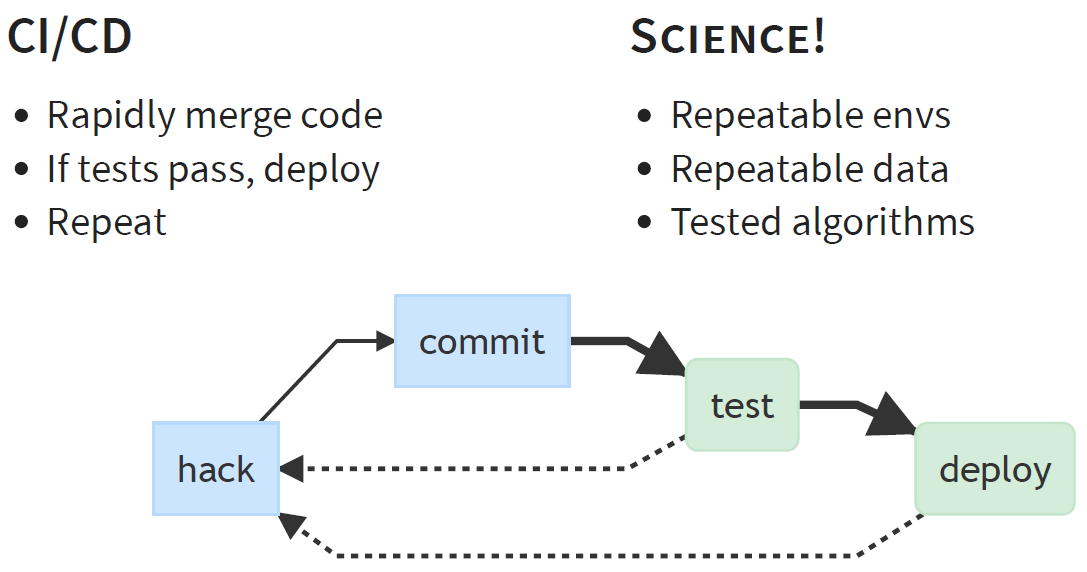
# PYTHON @ HIGHER LEVEL



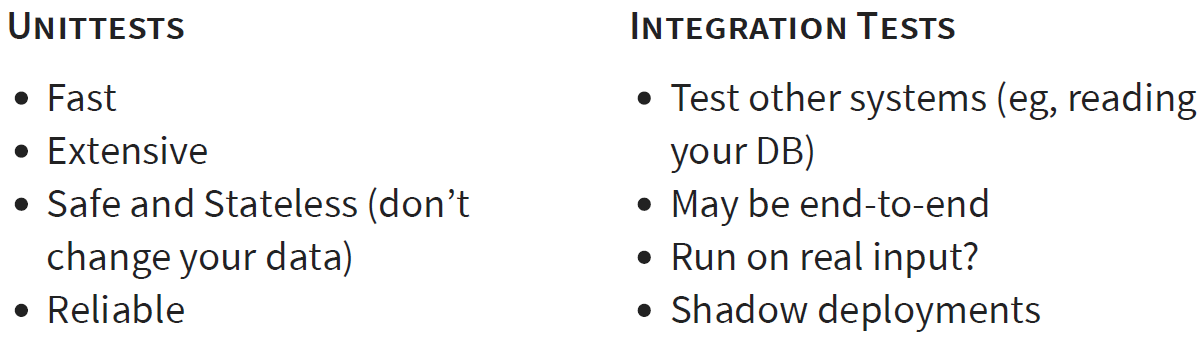
# REPEATABLE ENVIRONMENT



# CI/CD



# Testing

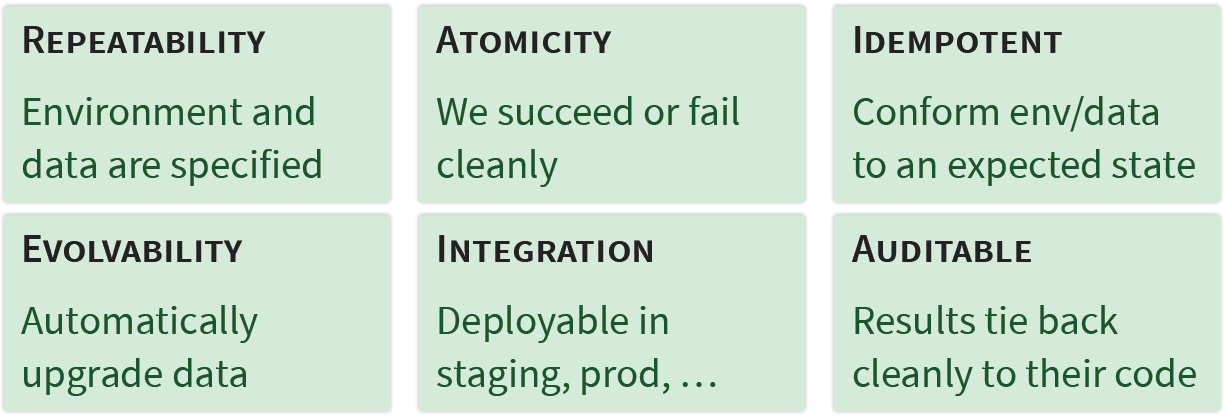


But Science is special

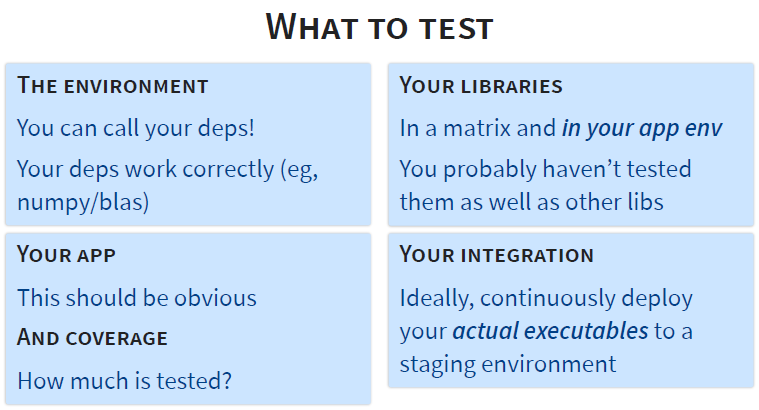
|  |  |
| --- | --- |
| * More complicated environments * Stateful, historical data * Persisted results * Probabilistic algorithms (random failures) | * ‘Qualitative’ results (eg, is model accuracy good enough?) * Evolving input data schemas * Hard to get representative data * Integrated, not isolated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

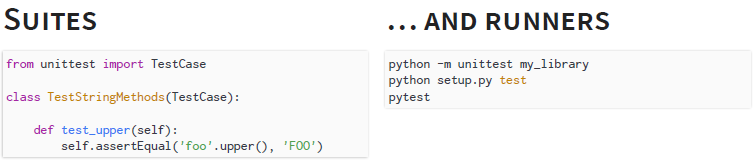
Therefore ….

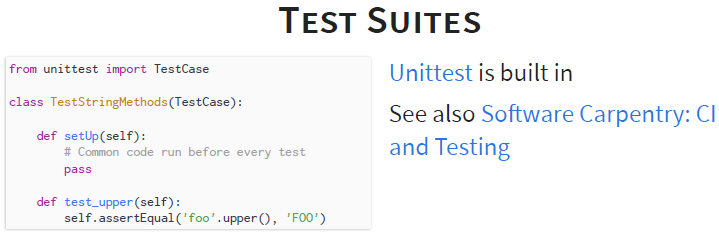
# STRATEGY for Data Science



# 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)

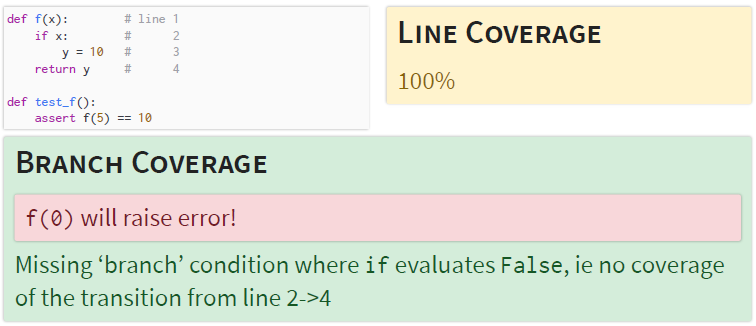
**def** **test\_ehlo**(self, smtp\_connection=**None**):

response, msg = smtp\_connection.ehlo()

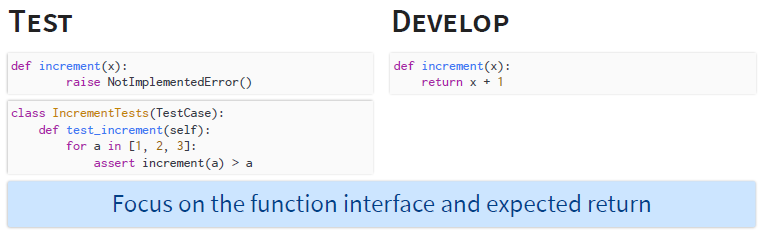
...

## PYTEST PLUGINS Coverage

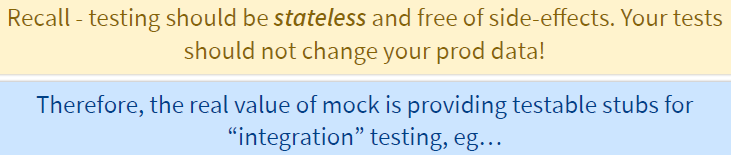
Branch coverage is far more important than line coverage. It looks for tests on every logical branch code could follow.



## 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')

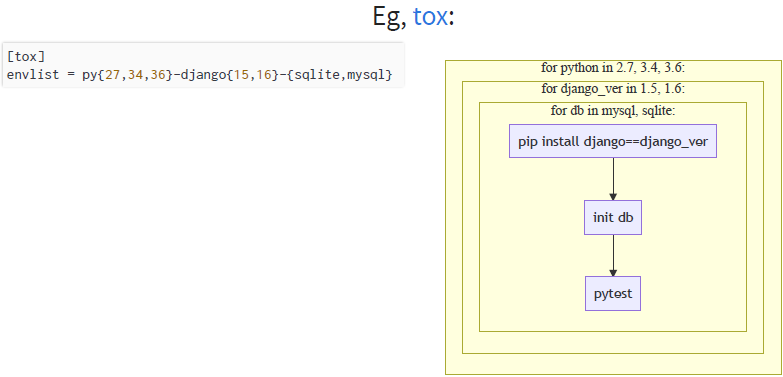
model\_instance.save()

**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'

)

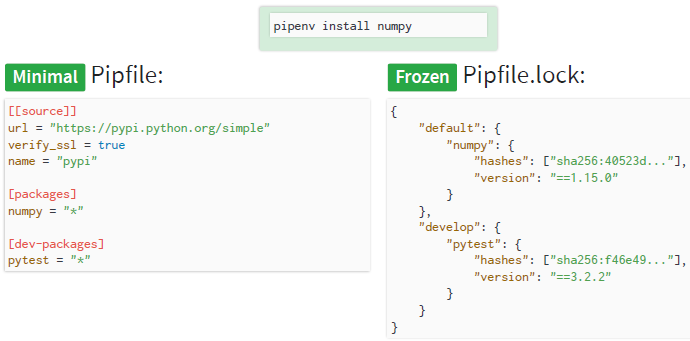
)

Not all configs are secrets. Distinguish between ‘**deploy configs**’ (points to servers, files) and ‘**code configs**’ (runs with verbose debugging, or higher training epochs). For the latter, it makes sense to version control the config (or set profiles in code, etc) since they are likely shareable between deploys.

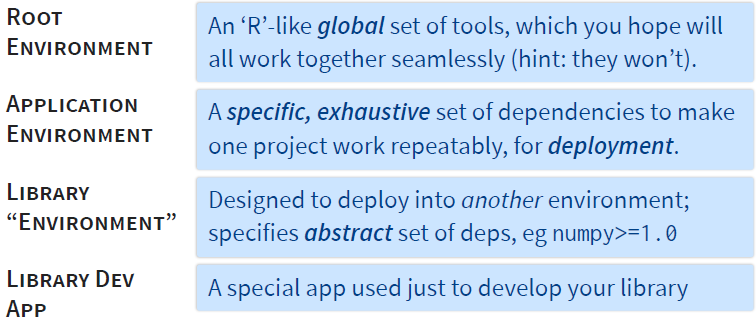
Can use a **.env** file or **env directory** to specify env variables. **Pipenv** and **docker** automatically read these, and other tools like **envdir** can load them explicitly

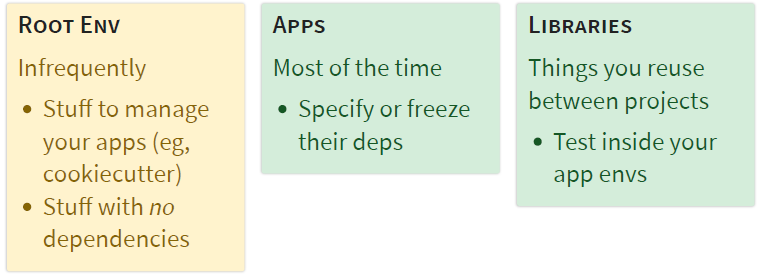
Usually these files should not be in VCS! Other tools prefer **INI** or **yaml** systems - eg python’s built in **ConfigParser** and **Luigi**: great composability (eg a system file for servers/IP’s, local file for code config in VCS) and even variables in the configs BUT often not overrideable via the environment.

# PIPENV

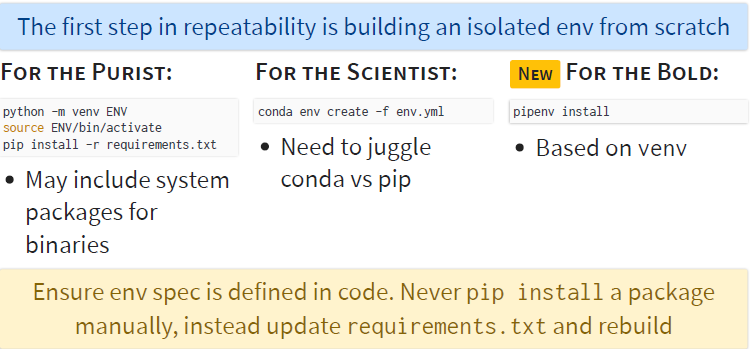


## Environments, Apps, Libraries

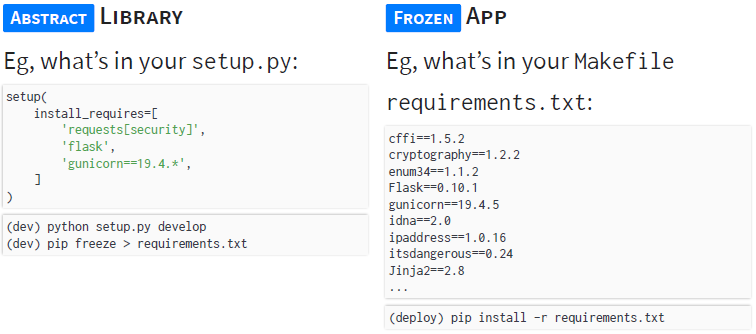




## Virtual Environments



## What are your REQS?



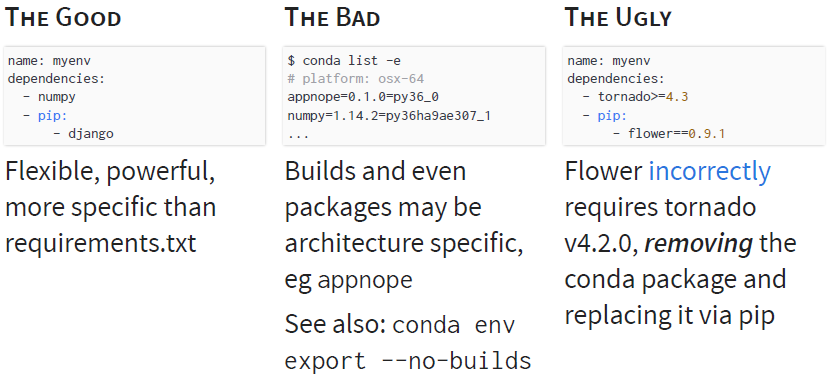
## Why PIPENV?

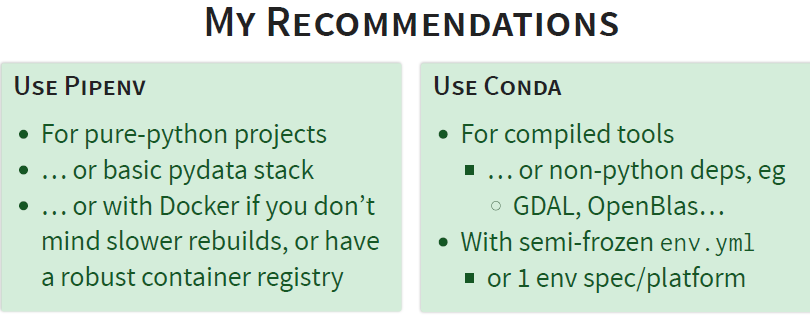


pipenv update

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

## CONDA

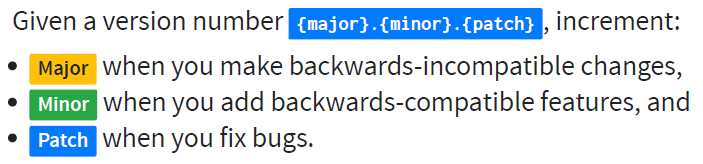




## Libraries versus Applications

* 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

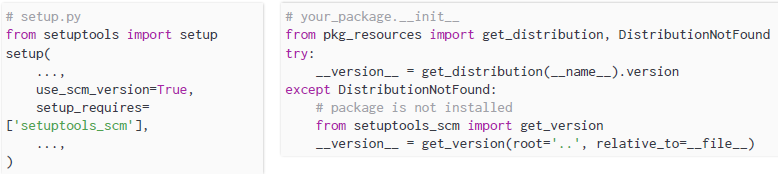
* Need to find a way to specify version once and get it right
* Need version to be globally unique
* Need to account for dev versions for your own code

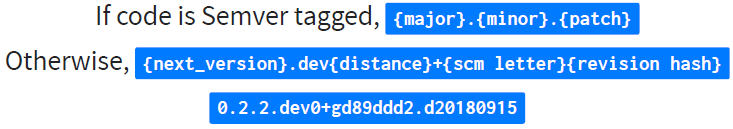
Simple: specify via hardcode in version.py and import in setup.py. But:

* 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

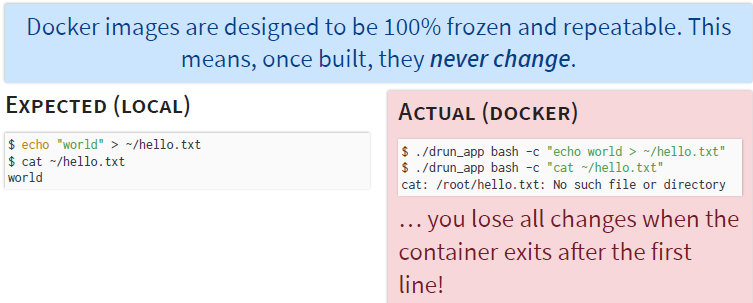
## SETUPTOOLS\_SCM

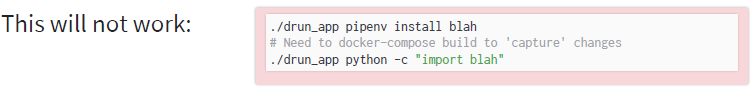
Manage your versions by scm tags



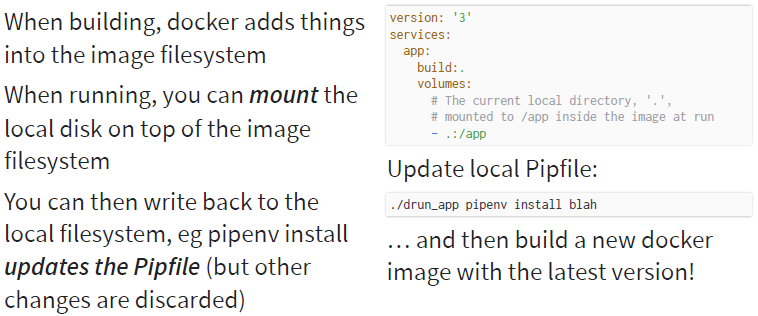


# DOCKER

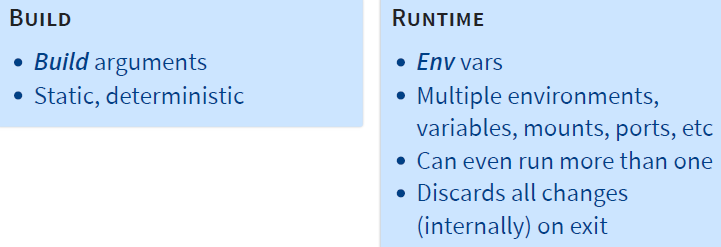


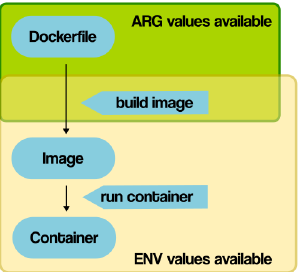


## Build vs. Mounting



## 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**

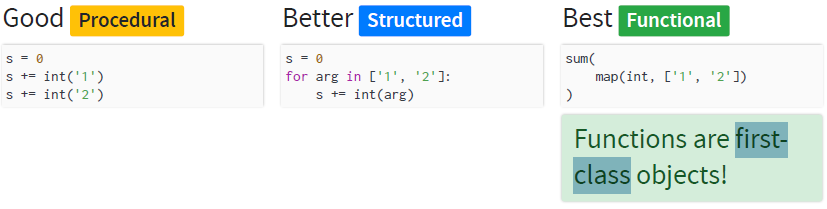
**from** **logging** **import** root

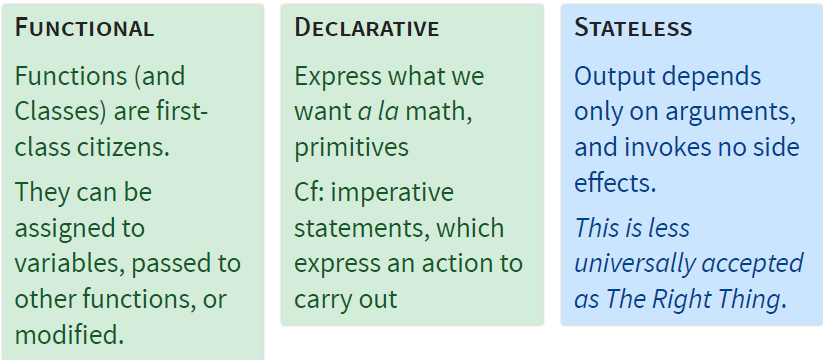
**import** **my\_app**

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

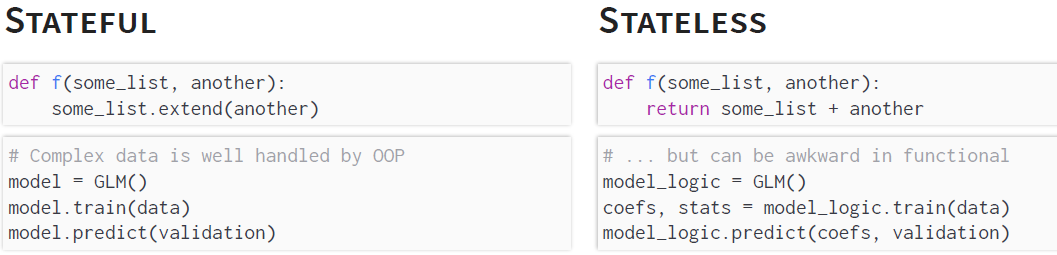
# Functional Programming

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

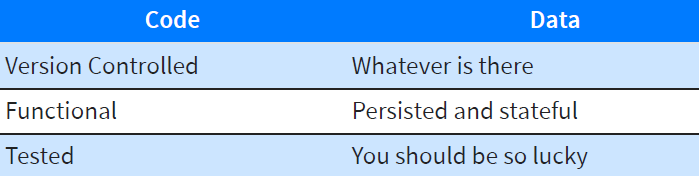




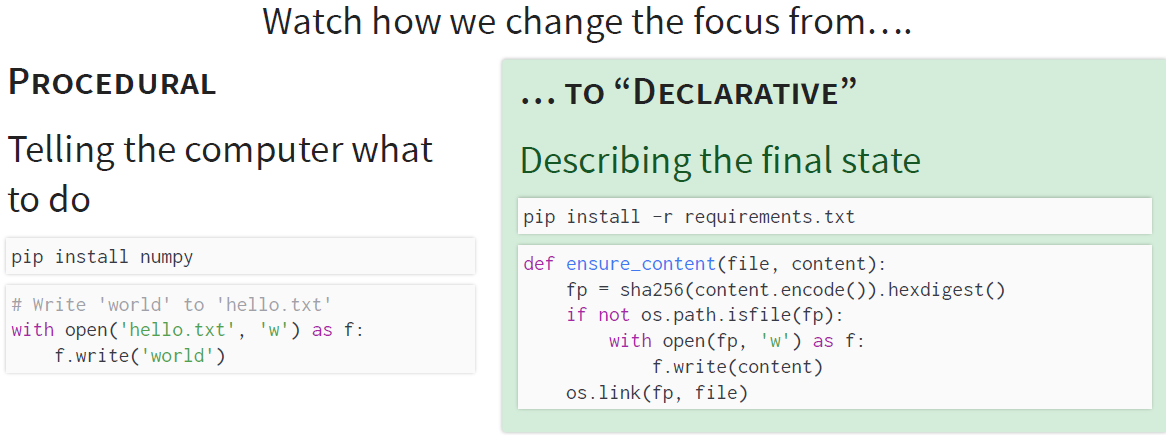
## STATE



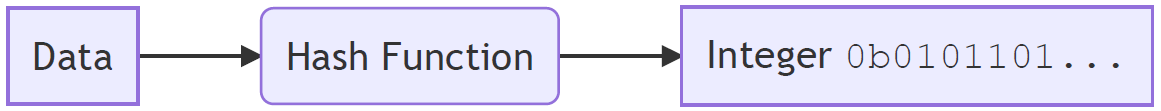
**Sorry State of Stateful Data! Changes in code not reflected in stateful data**



## DECLARATIVE vs. PROCEDURAL



# HASH

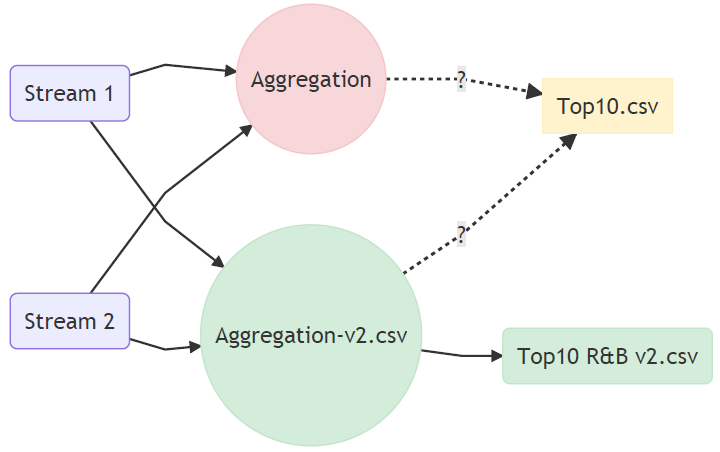


🞟**Hash function**, e.g. sha256, maps arbitrary string of data into fixedwidth integer, e.g. 256 bits 🞟Integer appears uniform and is deterministic 🢥 useful for many algorithms… 🞟Hashes distribute lookups evenly

# 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}'

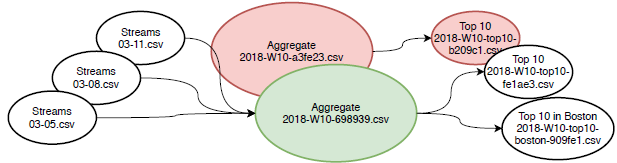
**for** pname, pvalue **in** task.param\_kwargs.items()

]

)

**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.

You call the task on the input, and hash the 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()

# 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

# LUIGI

## Atomicity

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

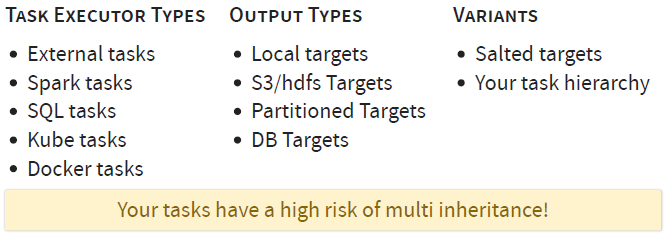
* Define .write(), .exists() atomically for a target
  + May never return True for an incomplete or failed output
* A task is runnable when all input targets .exists()
* A task is done when it’s output target .exists()

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

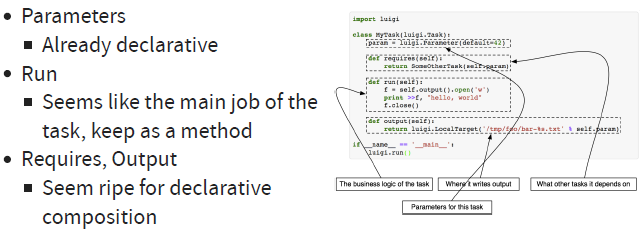
|  |  |
| --- | --- |
|  | Tasks .require() other Tasks, forming a DAG (they don’t require Targets!) |

## 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()

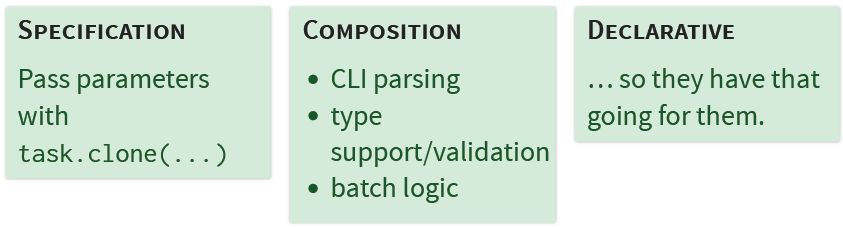
artist\_count[artist] += **1**

**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:

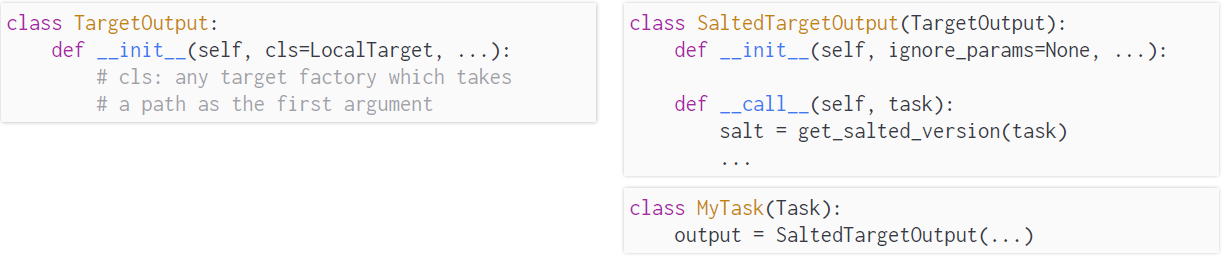
(1) Any relationship between file path & task name? eg, w/ configurable root directory {DATA\_ROOT}/{param}/{task\_name}.txt (2) Is output salted?

(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* |

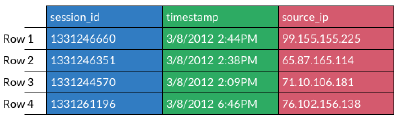
Two compositional tools in Luigi Utils – Work for basic param sharing

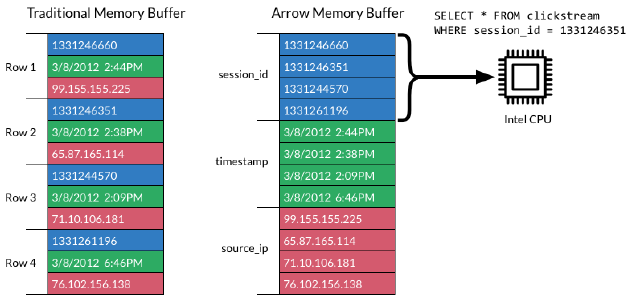
|  |  |
| --- | --- |
| *# Copies parameters*  @inherits(TaskA)  **class** TaskB:  ... | *# Copies params and requires*  @requires(TaskB)  **class** TaskC(Task):  ... |

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



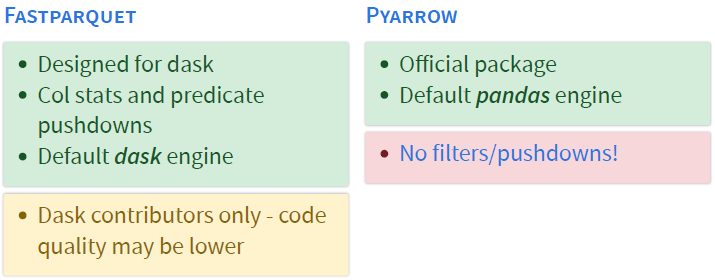


|  |  |
| --- | --- |
| A picture containing screenshot  Description automatically generated | **Apache Parquet**  Emerging standard for distributed,  columnar binary data Statistics for predicate pushdown  <http://parquet.apache.org/>  **Statistics**  Parquet stores partition  min/max values to allow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

## Column Stores

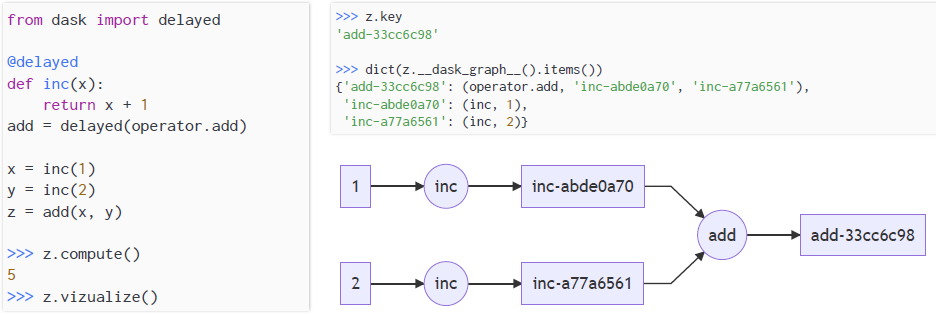
|  |  |
| --- | --- |
| A screenshot of a cell phone  Description automatically generated | Most operations only care about a few columns. Don’t read them all if you don’t need to! |
| A screenshot of a cell phone  Description automatically generated | Column stores offer superior:   * Compression, due 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 | | |  | | |
| 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**

# 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)

### 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

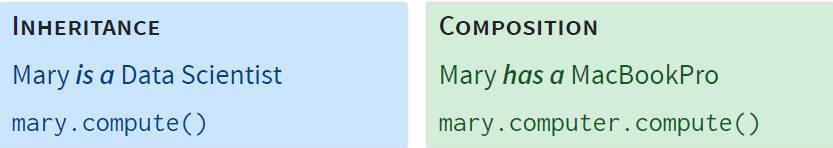
**class** **JSONView**(JSONResponseMixin, TemplateView):

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

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

## Composition 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  # Invoked via `A().x`  **return** obj.\_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!* |

# 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