noWorkflow: a Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts

João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, Juliana Freire

Agenda

- Background and Important Concepts
- Overview of noWorkflow (Murta, et. al 2014)
- Demonstration
    - Provenance Collection
    - Definition Provenance
    - Deployment Provenance
    - Execution Provenance
- Provenance Analysis
- Provenance Management
- Conclusion

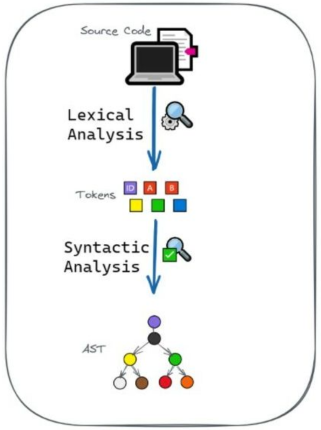
(Murta, et. al 2014) noWorkflow: Capturing and Analyzing Provenance of Scripts

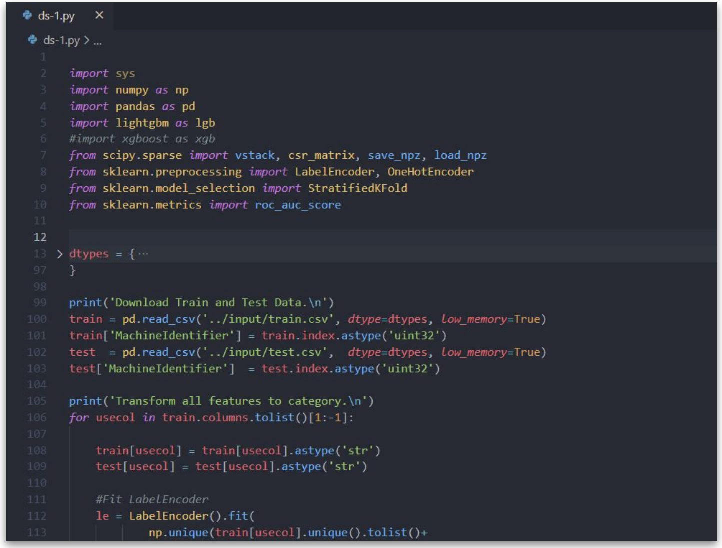
Background

- Provenance, scientific **reproducibility** and **evolution** of experiments.
- Previous works:
    - Operating system level: general but difficult to reason
    - Workflow management systems (WFMS): closely match experiment semantics but high adoption costs
    - Current script approaches: do not support repeatability and experiment evolution
- noWorkflow:
    - Provenance from Python scripts
    - Tracking history and evolution
    - Analysis of multiple trials
    - Prospective provenance (from YesWorkflow) + retrospective provenance (noWorkflow)

Important concepts

- **Static analysis**
  - Examining code without executing
  - Type checking
  - Control flow analysis (execution paths)
  - Dataflow analysis (flow through variables and functions)
  - Conservative (consider worst-case scenarios)

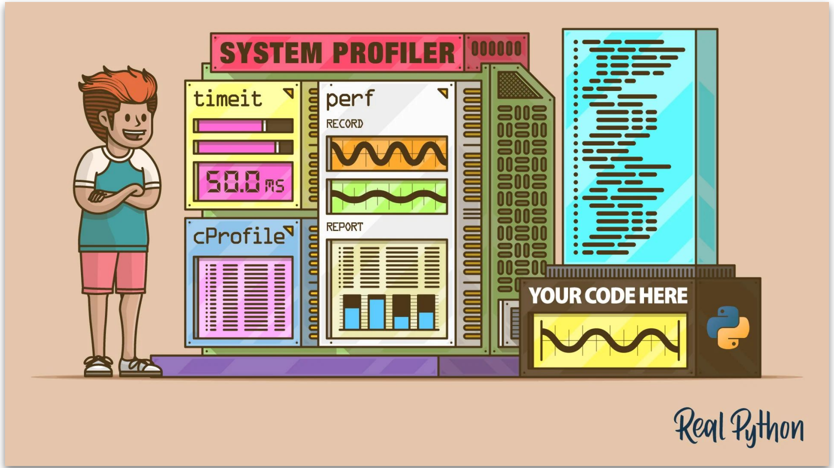




```python

import sys
import numpy as np
import pandas as pd
import lightgbm as lgb
#import xgboost as xgb
from scipy.sparse import vstack, csr_matrix, save_npz, load_npz
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import roc_auc_score


dtypes = {···
}

print('Download Train and Test Data.\n')
train = pd.read_csv('../input/train.csv', dtype=dtypes, low_memory=True)
train['MachineIdentifier'] = train.index.astype('uint32')
test  = pd.read_csv('../input/test.csv',  dtype=dtypes, low_memory=True)
test['MachineIdentifier']  = test.index.astype('uint32')

print('Transform all features to category.\n')
for usecol in train.columns.tolist()[1:-1]:

    train[usecol] = train[usecol].astype('str')
    test[usecol] = test[usecol].astype('str')

    #Fit LabelEncoder
    le = LabelEncoder().fit(
            np.unique(train[usecol].unique().tolist()+
```

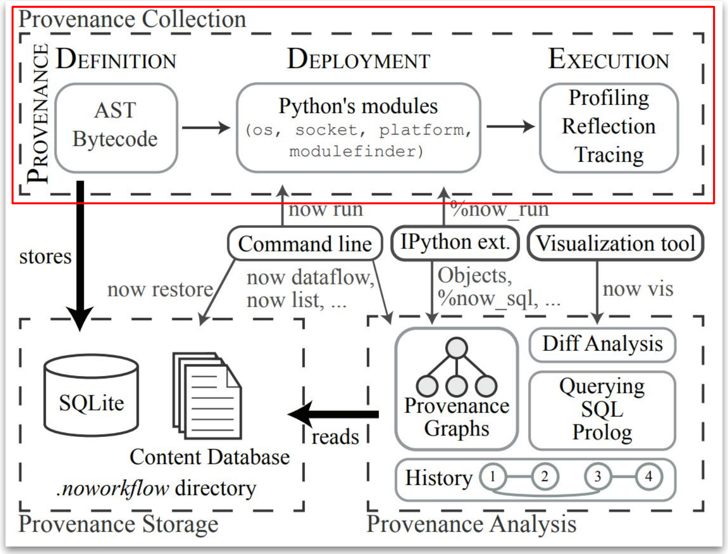
Important concepts

- **Runtime monitoring**
  - Expanding static analysis
  - Use of network
  - Detecting anomalies (e.g., memory usage)
  - Ensuring security (e.g., no unauthorized access)
  - Profilers

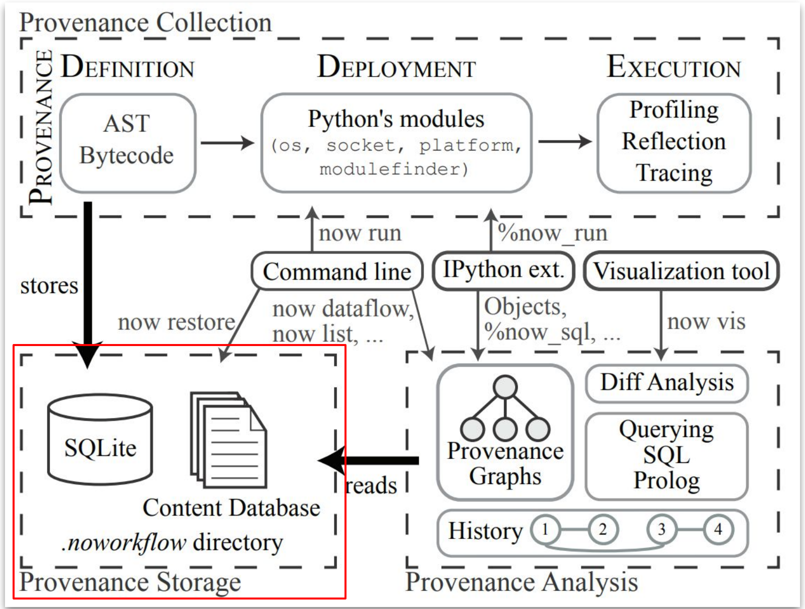


Overview of noWorkflow

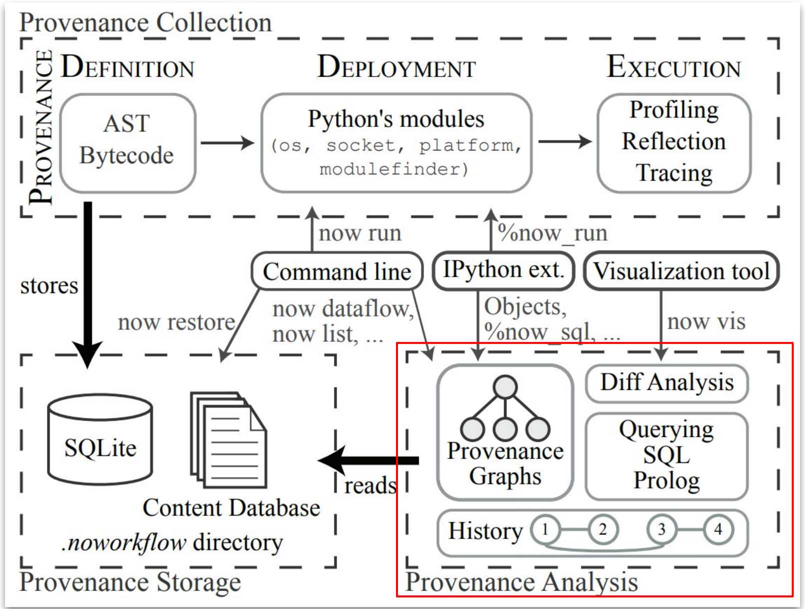
- Appropriate level of granularity
- Difficult to determine which parts of script produced data



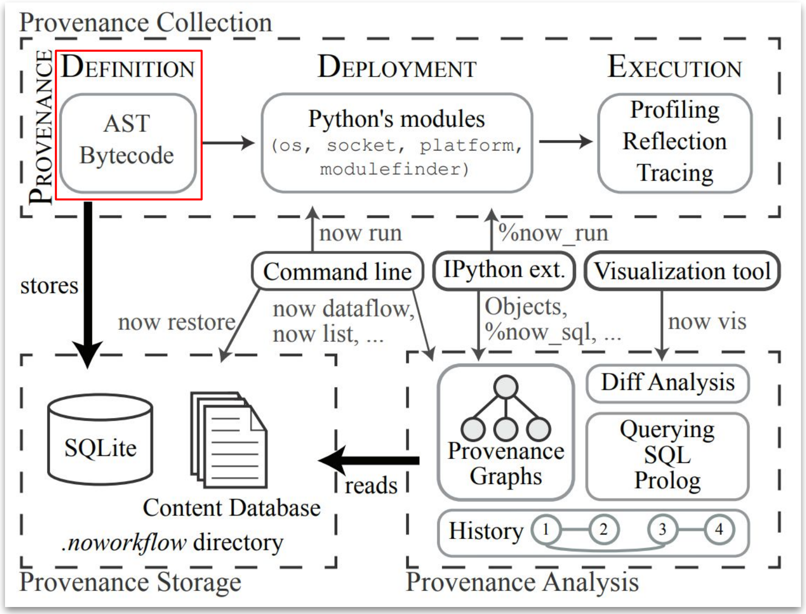
Overview of noWorkflow



Overview of noWorkflow

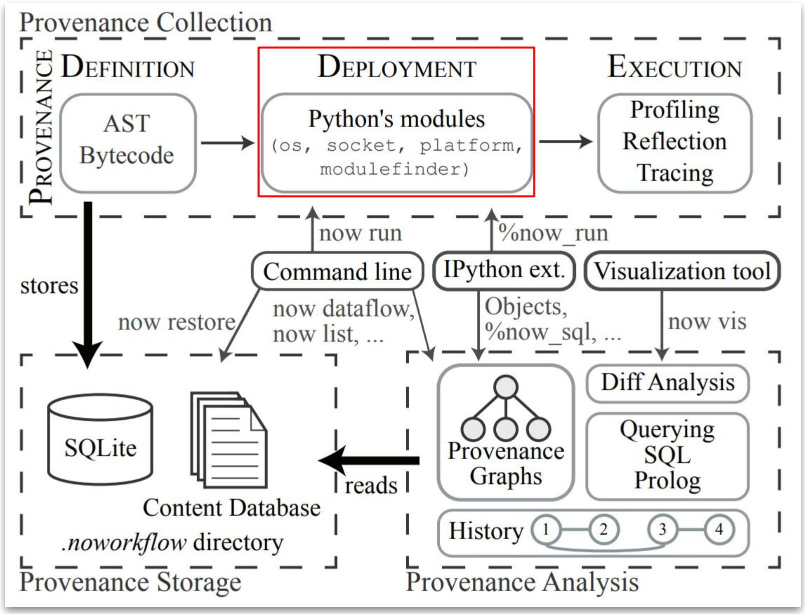


Overview of noWorkflow



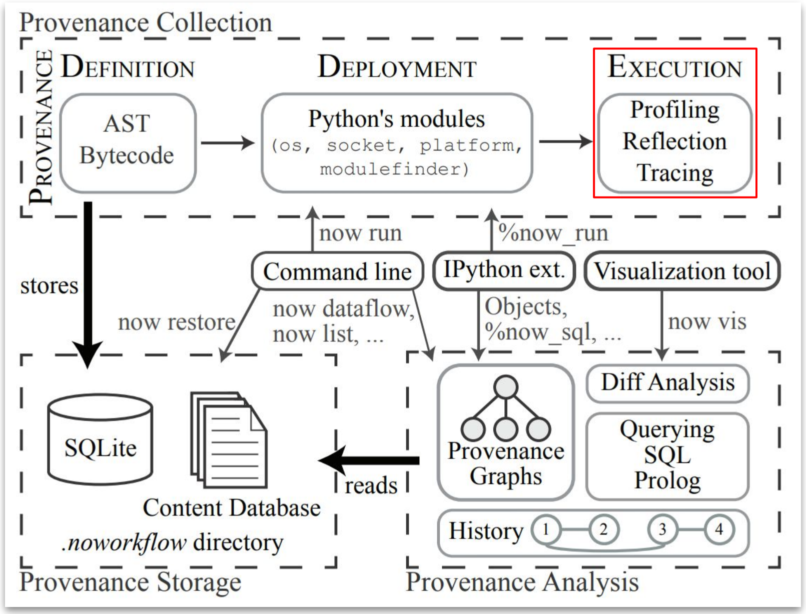
Definition provenance: "represents the structure of the script, including function definitions, their arguments, function calls, and other static data".

Overview of noWorkflow



Deployment provenance: “represents the execution environment, including information about the operating system, environment variables, and libraries on which the script depends”.

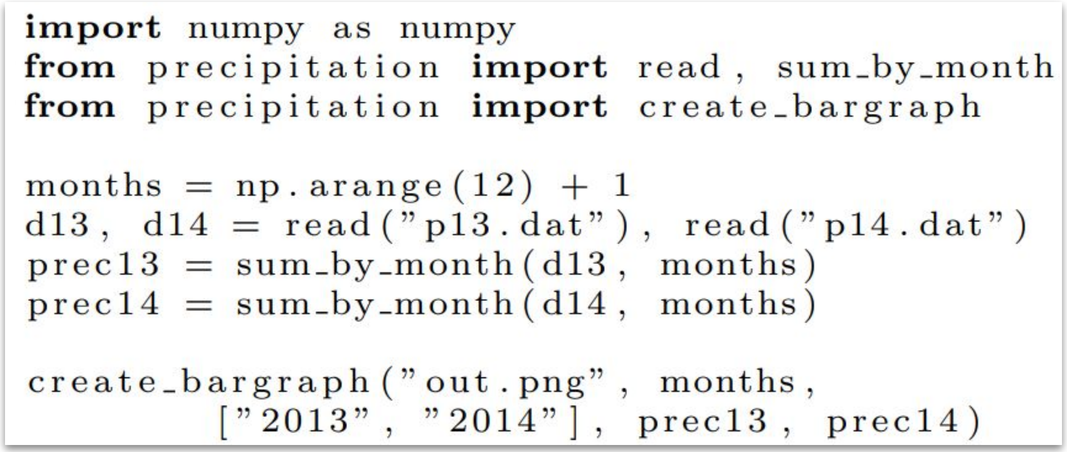
Overview of noWorkflow



Execution provenance: "represents the execution log for the script".

Demonstration

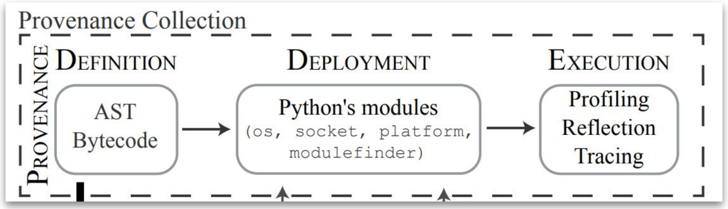
- Checking if the **precipitation** in Rio de Janeiro remains constant **across years** (2013 and 2014).
- **Collecting** data from meteorological database, **process** the data and **produce** an image for comparison.



```
import numpy as numpy
from precipitation import read, sum_by_month
from precipitation import create_bargraph

months = np.arange(12) + 1
d13, d14 = read(”p13.dat”), read(”p14.dat”)
prec13 = sum_by_month(d13, months)
prec14 = sum_by_month(d14, months)

create_bargraph(”out.png”, months,
        [”2013”, ”2014”], prec13, prec14)
```

Provenance Collection

- Attribute trial number
- 1) **definition provenance**, 2) **deployment provenance.** After execution 3) **execution provenance**
- Accessed files, modules, and scripts stored to files. SHA1 hash to files stored on the database

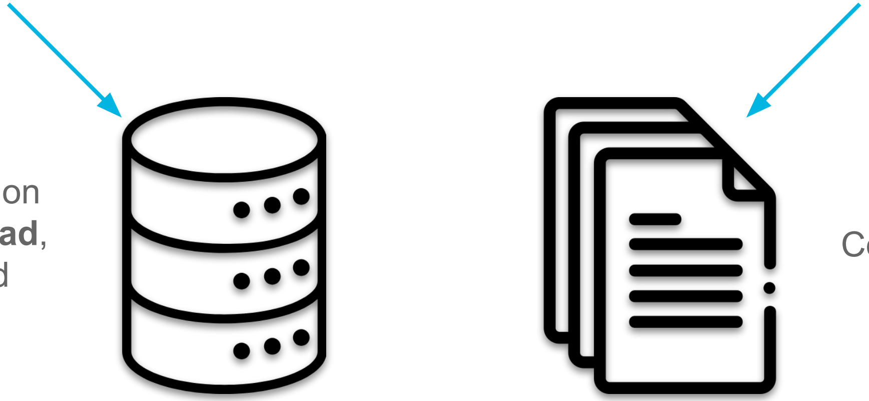


Definition Provenance

- Prospective provenance
- Abstract Syntax Tree (AST)
- Python bytecode of the script

function **names**, **calls**, **parameters**, and **global variables**

script file and **function definition**



Hash code and function calls (i.e., **arange**, **read**, **sum_by_month**, and **create_bargraph**)

Content of ‘*experiment.py*’

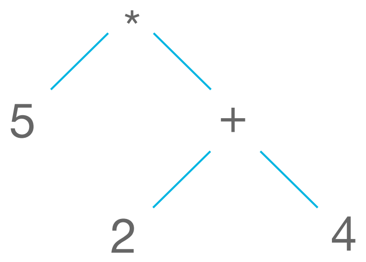
Relational Database

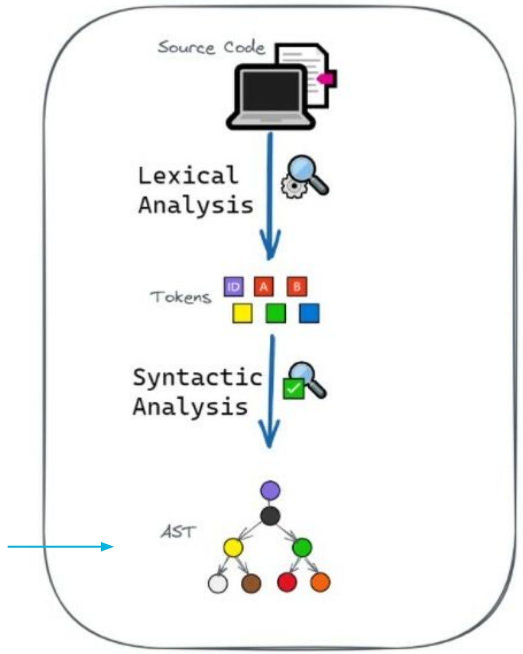
Content Database

Abstract Syntax Tree (AST)

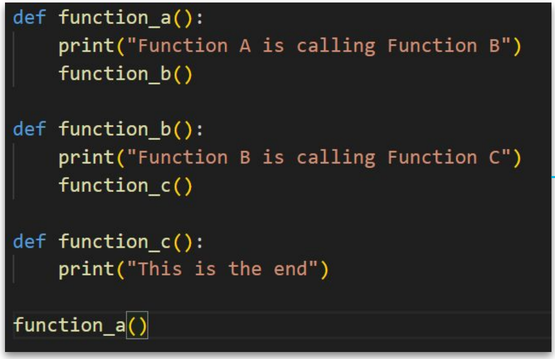
- AST models relationship between tokens as a tree of nodes containing children. Each node contains type of token and related data.

5 * (2 + 4)

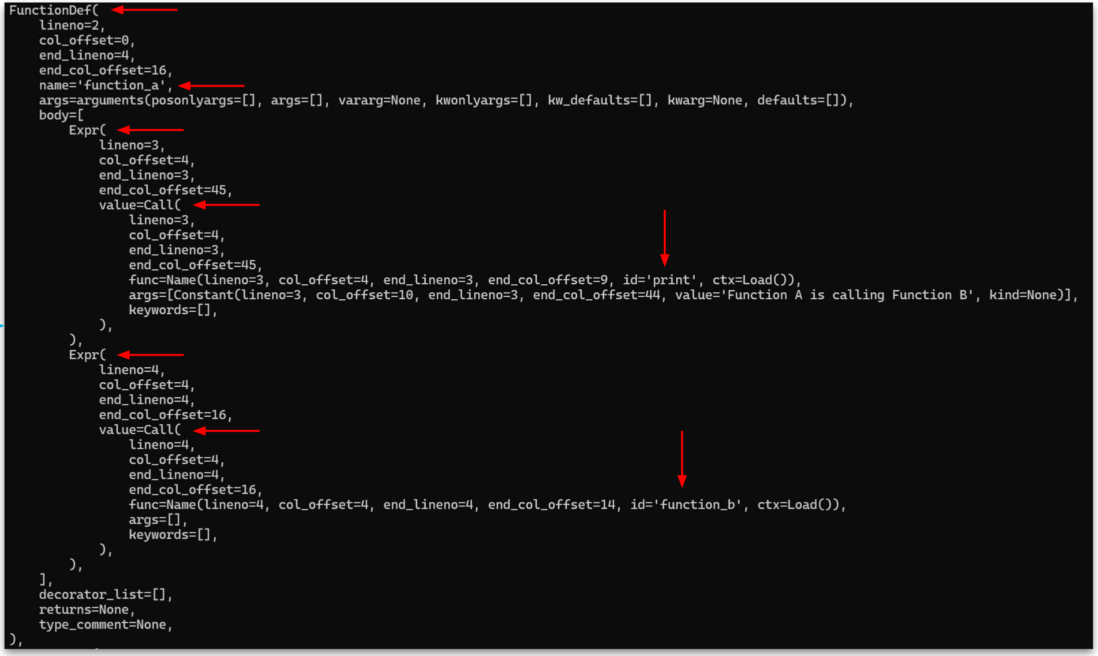




Abstract Syntax Tree (AST)

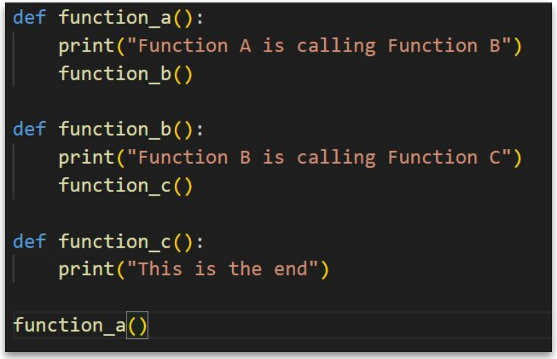


```python
def function_a():
    print("Function A is calling Function B")
    function_b()

def function_b():
    print("Function B is calling Function C")
    function_c()

def function_c():
    print("This is the end")

function_a()
```

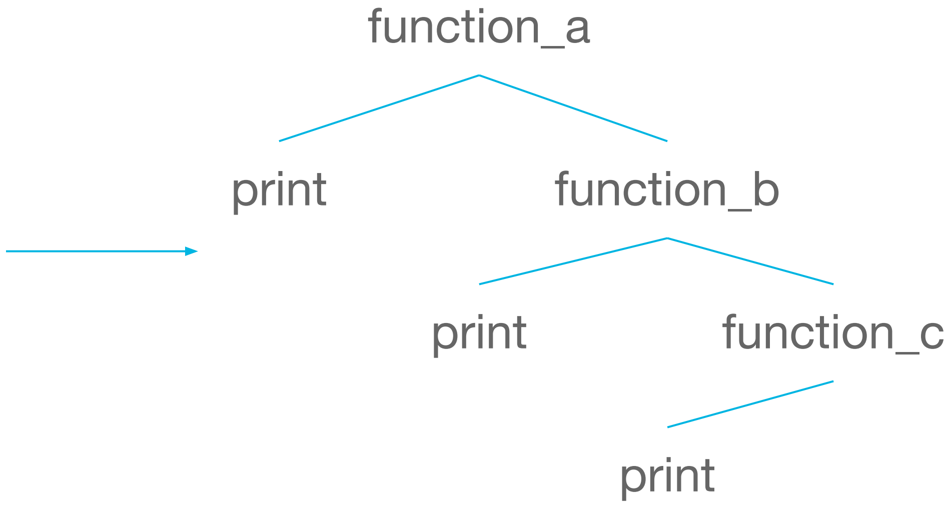


```
FunctionDef(
    lineno=2,
    col_offset=0,
    end_lineno=4,
    end_col_offset=16,
    name='function_a',
    args=arguments(posonlyargs=[], args=[], vararg=None, kwonlyargs=[], kw_defaults=[], kwarg=None, defaults=[]),
    body=[
        Expr(
            lineno=3,
            col_offset=4,
            end_lineno=3,
            end_col_offset=45,
            value=Call(
                lineno=3,
                col_offset=4,
                end_lineno=3,
                end_col_offset=45,
                func=Name(lineno=3, col_offset=4, end_lineno=3, end_col_offset=9, id='print', ctx=Load()),
                args=[Constant(lineno=3, col_offset=10, end_lineno=3, end_col_offset=44, value='Function A is calling Function B', kind=None)],
                keywords=[],
            ),
        ),
        Expr(
            lineno=4,
            col_offset=4,
            end_lineno=4,
            end_col_offset=16,
            value=Call(
                lineno=4,
                col_offset=4,
                end_lineno=4,
                end_col_offset=16,
                func=Name(lineno=4, col_offset=4, end_lineno=4, end_col_offset=14, id='function_b', ctx=Load()),
                args=[],
                keywords=[],
            ),
        ),
    ],
    decorator_list=[],
    returns=None,
    type_comment=None,
),
```

Abstract Syntax Tree (AST)

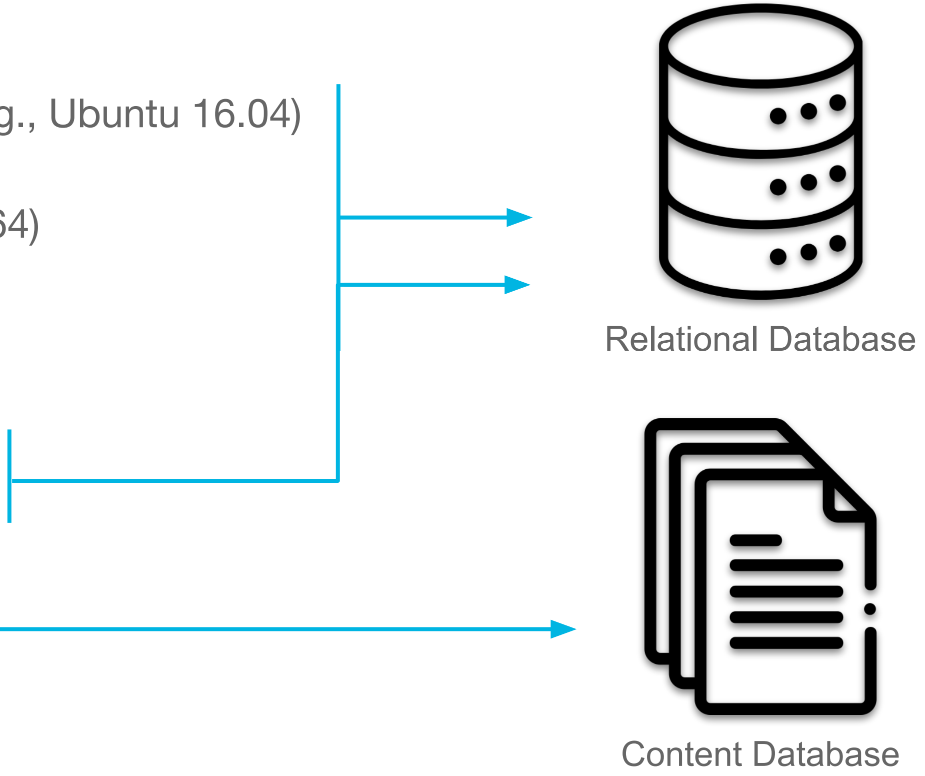


```
def function_a():
    print("Function A is calling Function B")
    function_b()

def function_b():
    print("Function B is calling Function C")
    function_c()

def function_c():
    print("This is the end")

function_a()
```



Deployment Provenance

- Environment
  - Operating system information (e.g., Ubuntu 16.04)
  - Hostname
  - Machine Architecture (e.g., x86_64)
  - Python version (e.g., 3.5.2)
  - Environment variables
- Library dependencies
  - Versions (e.g., ‘numpy’ in 1.11.3)
  - Names
  - Transitive closure
  - Libraries



Execution Provenance

Retrospective provenance and **Runtime monitoring**

- **Input** and **output** files before and after processing (e.g., ‘p13.dat’ and ‘output.png’)

Content Database

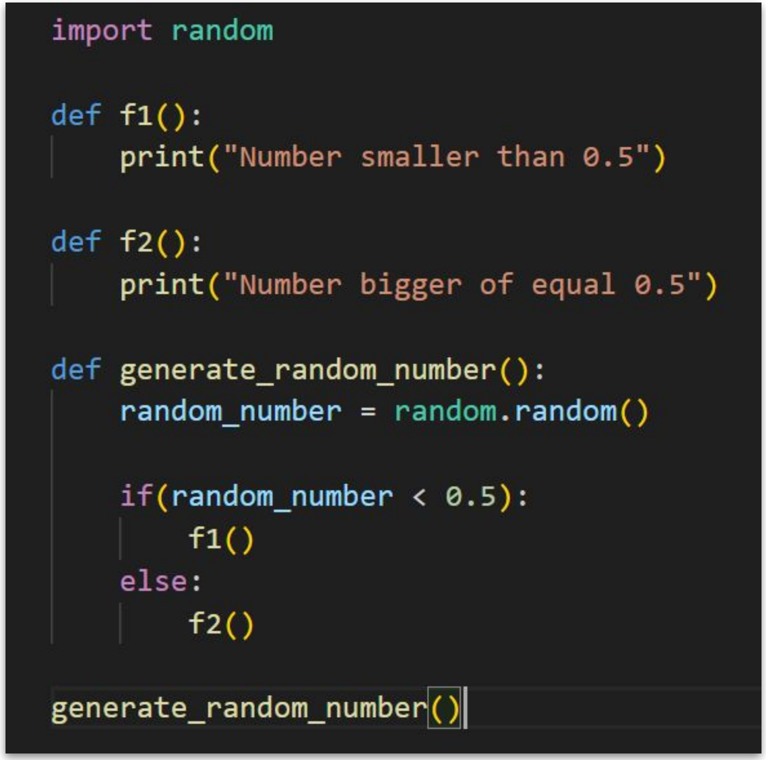
- Two granularities:
  - **Coarse** (Python Profiler): function activations (i.e., executed function calls), global variables, parameters, and return values
  - **Fine** (Profiler + Tracer): variable attributions, loop definitions, variable dependencies

Relational Database

Execution Provenance

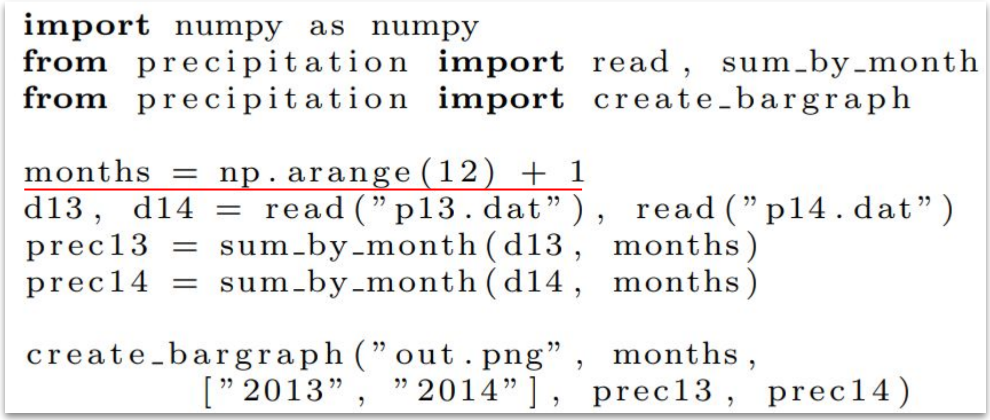
Function call: related to definition provenance. Can be captured by static analysis.

Function activation: related to execution provenance. Only captured in runtime.



```python
import random

def f1():
    print("Number smaller than 0.5")

def f2():
    print("Number bigger of equal 0.5")

def generate_random_number():
    random_number = random.random()

    if(random_number < 0.5):
        f1()
    else:
        f2()

generate_random_number()
```

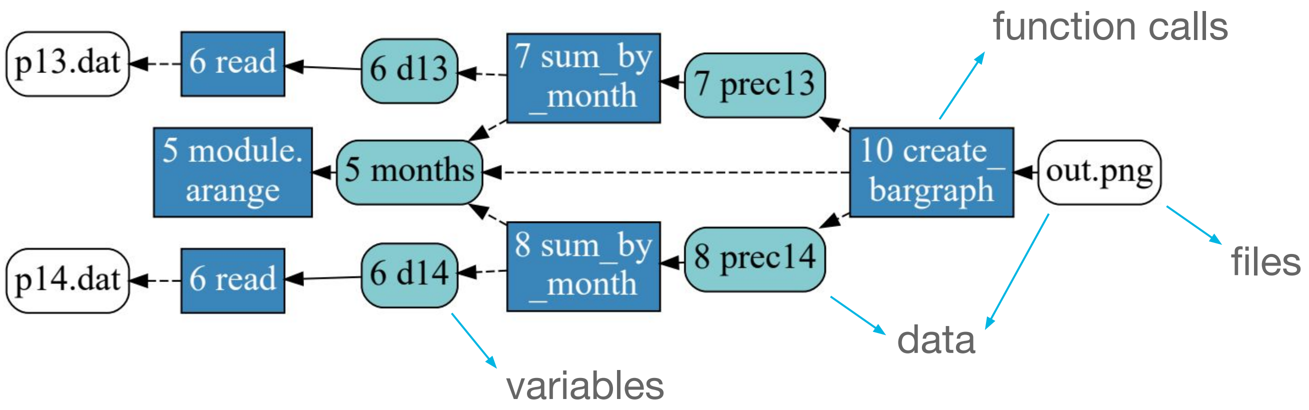
Execution Provenance (Example)


```
import numpy as numpy
from precipitation import read, sum_by_month
from precipitation import create_bargraph

months = np.arange(12) + 1
d13, d14 = read(”p13.dat”), read(”p14.dat”)
prec13 = sum_by_month(d13, months)
prec14 = sum_by_month(d14, months)

create_bargraph(”out.png”, months,
        [”2013”, ”2014”], prec13, prec14)
```

- Coarse: **np.arange(12)** returns **[0, 1, ..., 10, 11]**
- Fine: value of **months** as **[1, 2, ..., 11, 12]**

Provenance Analysis

now dataflow 1 | dot -Tpng -o p1.png



Provenance Analysis

- Query provenance with SQL
- Export provenance to Prolog facts and common queries
- Textual comparison between trials (**now diff 1 2**)

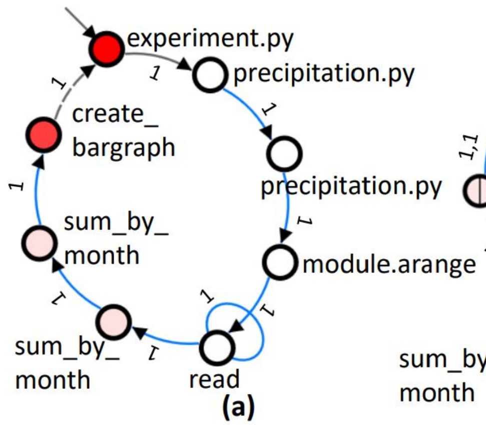
access_influence(1, File, ‘out.png’)

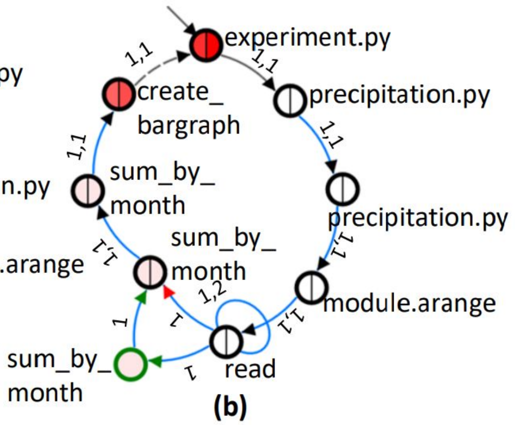
Which files might have influenced the generation of ‘output.png’ in trial 1

‘p13.dat’ and ‘p14.dat’

Provenance Analysis

- Web **visualization tool**
- History of trials as a graph



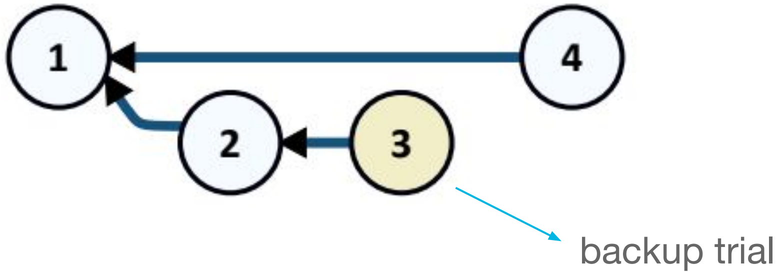


Activation graph of trial 1

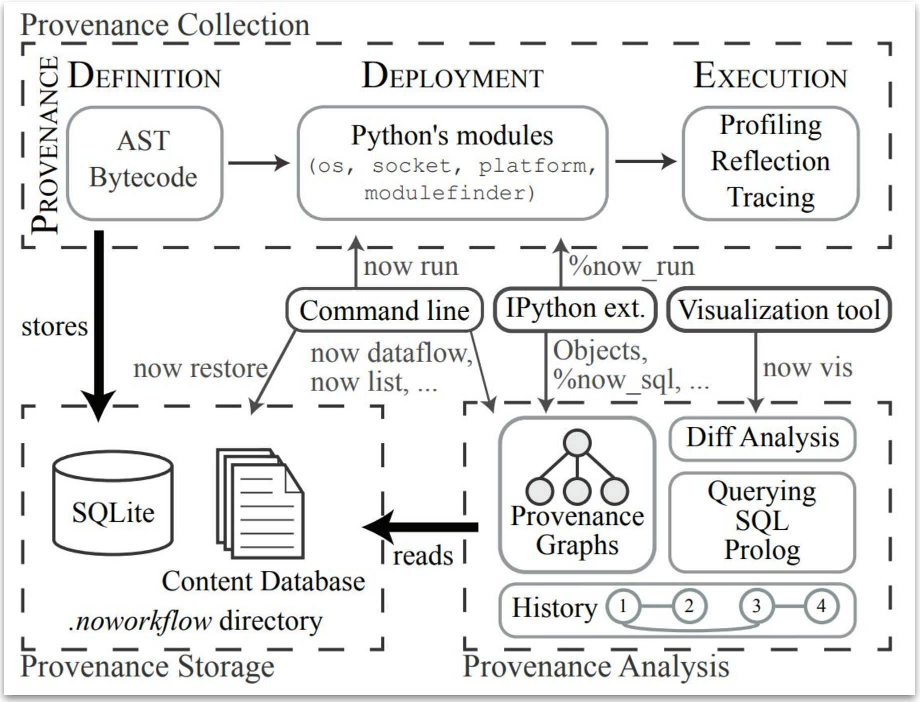
Diff to trial 2

Provenance Management

- Possible to restore code and data from previous trials
- Alternate scenarios
- Derivation history



Conclusion


Provenance Collection
PROVENANCE
DEFINITION
DEPLOYMENT
EXECUTION
AST
Bytecode
Python's modules
(os, socket, platform, modulefinder)
Profiling
Reflection
Tracing
now run
%now_run
stores
Command line
IPython ext.
Visualization tool
now restore
now dataflow, now list, ...
Objects, %now_sql, ...
now vis
SQLite
Content Database
.noworkflow directory
reads
Provenance Graphs
Diff Analysis
Querying
SQL
Prolog
History
1
2
3
4
Provenance Storage
Provenance Analysis

Conclusion

- Collecting provenance from Python scripts without modifying the script
- Tracking and navigating the evolution of experiments

Limitations:

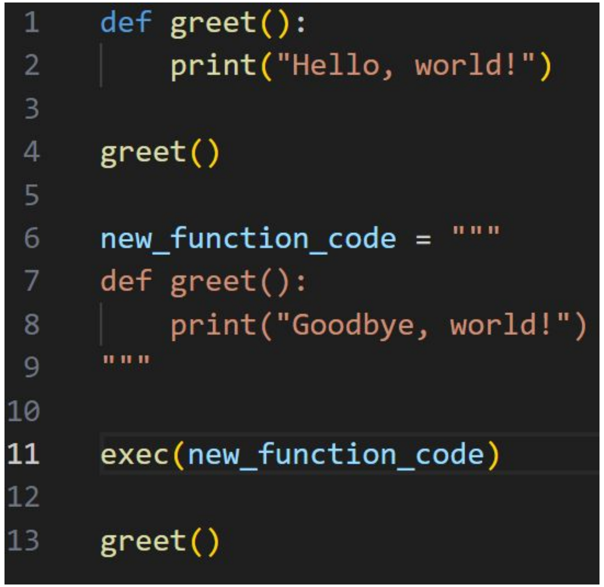
- Does not collect important data to some scripts (e.g., network or database)
- Only supports Python scripts
- Provenance size can grow with loops and size of scripts

Takeaways:

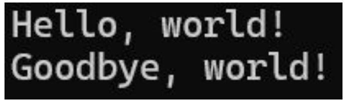
- The idea can be adapted to different languages and parts of the experiments
- Offers intuitive approach to provenance
- Calls for more initiatives that makes provenance accessible to final users

Conclusion

- Python is highly dynamic and unpredictable during runtime (no sound static analysis is possible in general case)



```python
def greet():
    print("Hello, world!")

greet()

new_function_code = """
def greet():
    print("Goodbye, world!")
"""

exec(new_function_code)

greet()
```



```
Hello, world!
Goodbye, world!
```

Thank you!