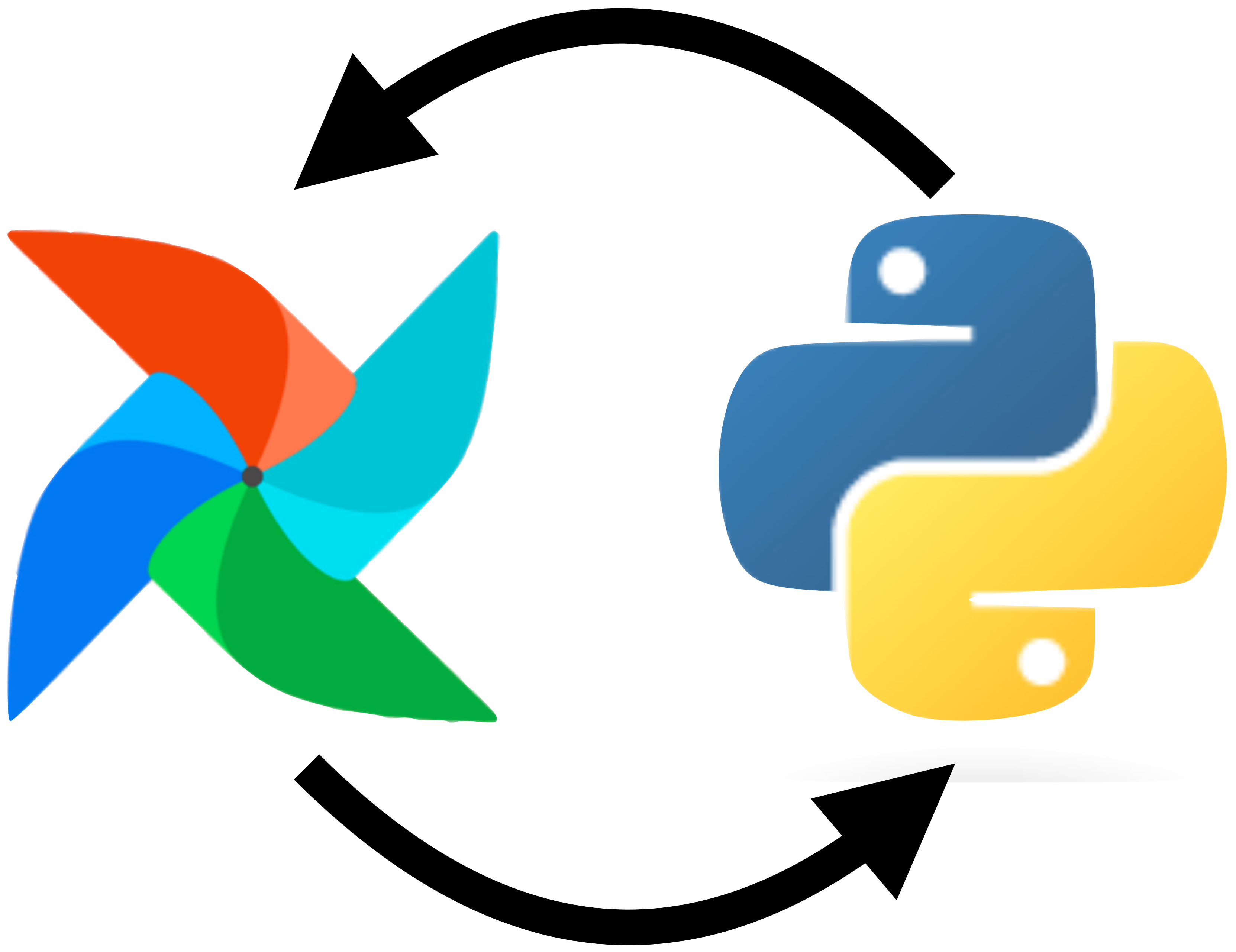
Workflows und Datentransformation mit Apache Airflow



Stefan Baerisch, stbaer.com, enterPy 2020

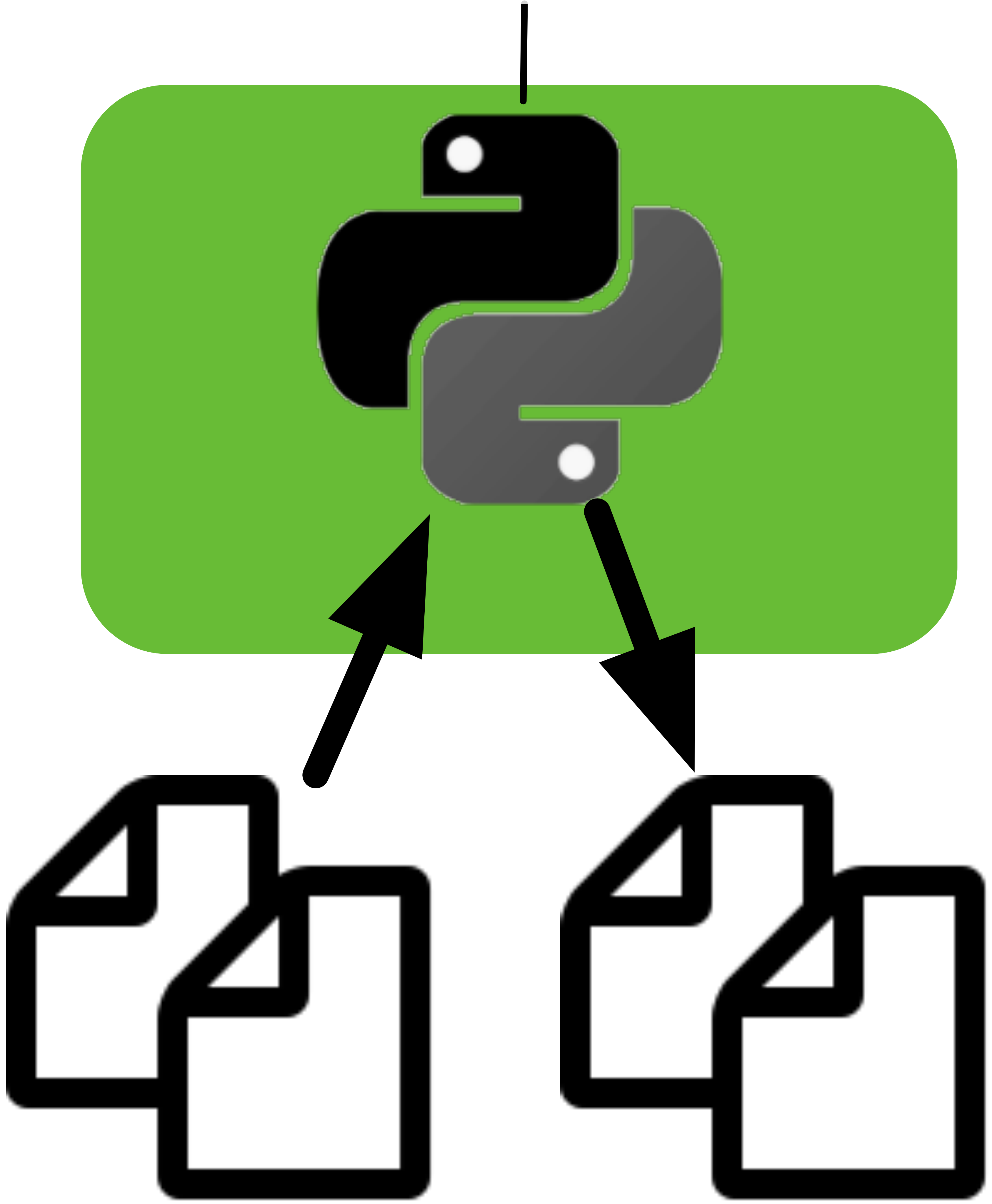
Stefan Baerisch

- Softwareentwickler und Projektmanager
- Python Anwender seit etwa 2008, Vorträge auf Konferenzen
- Python in diversen Projekten eingesetzt

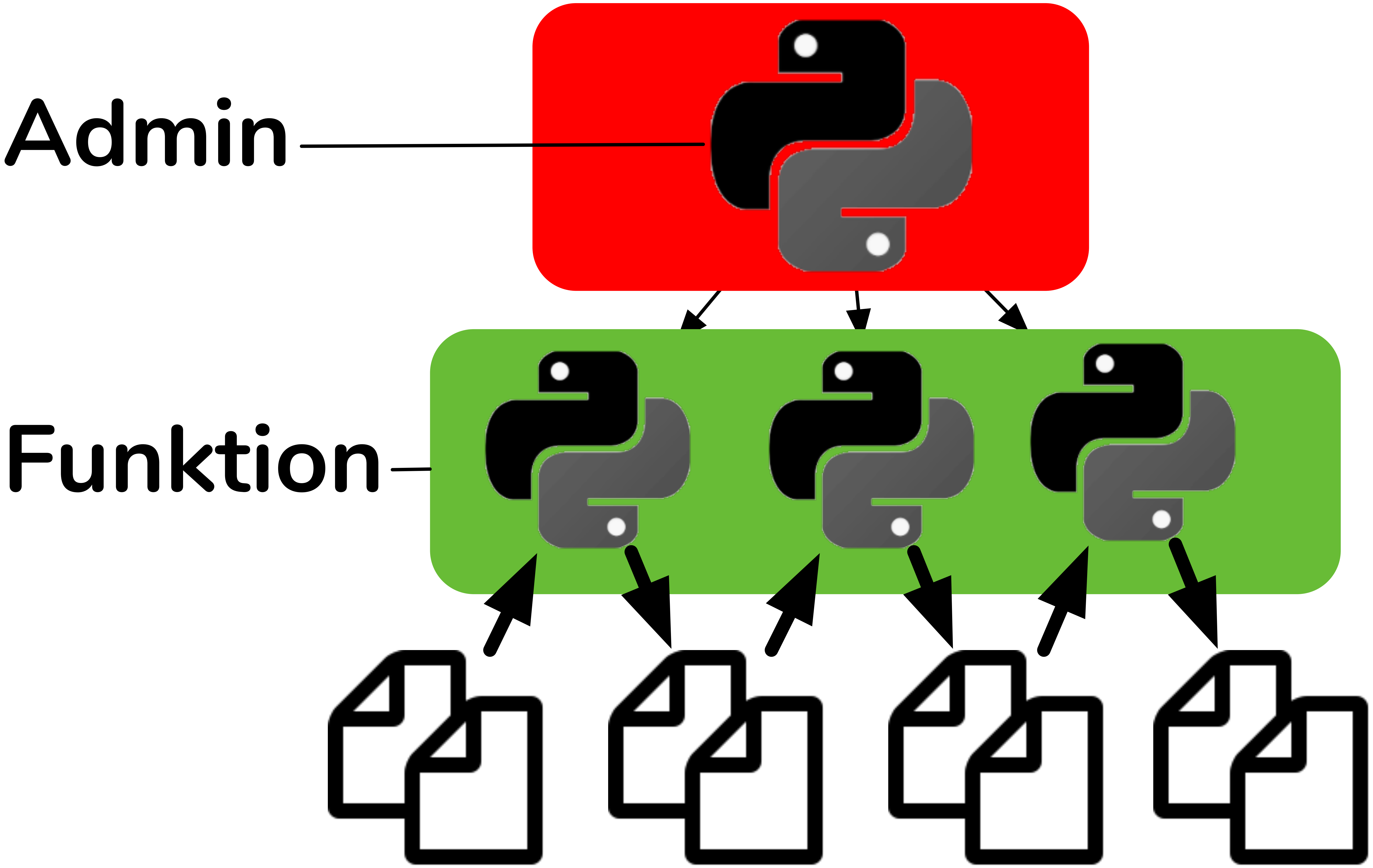
- https://stbaer.com

Warum dieser Vortrag (1)

Funktion

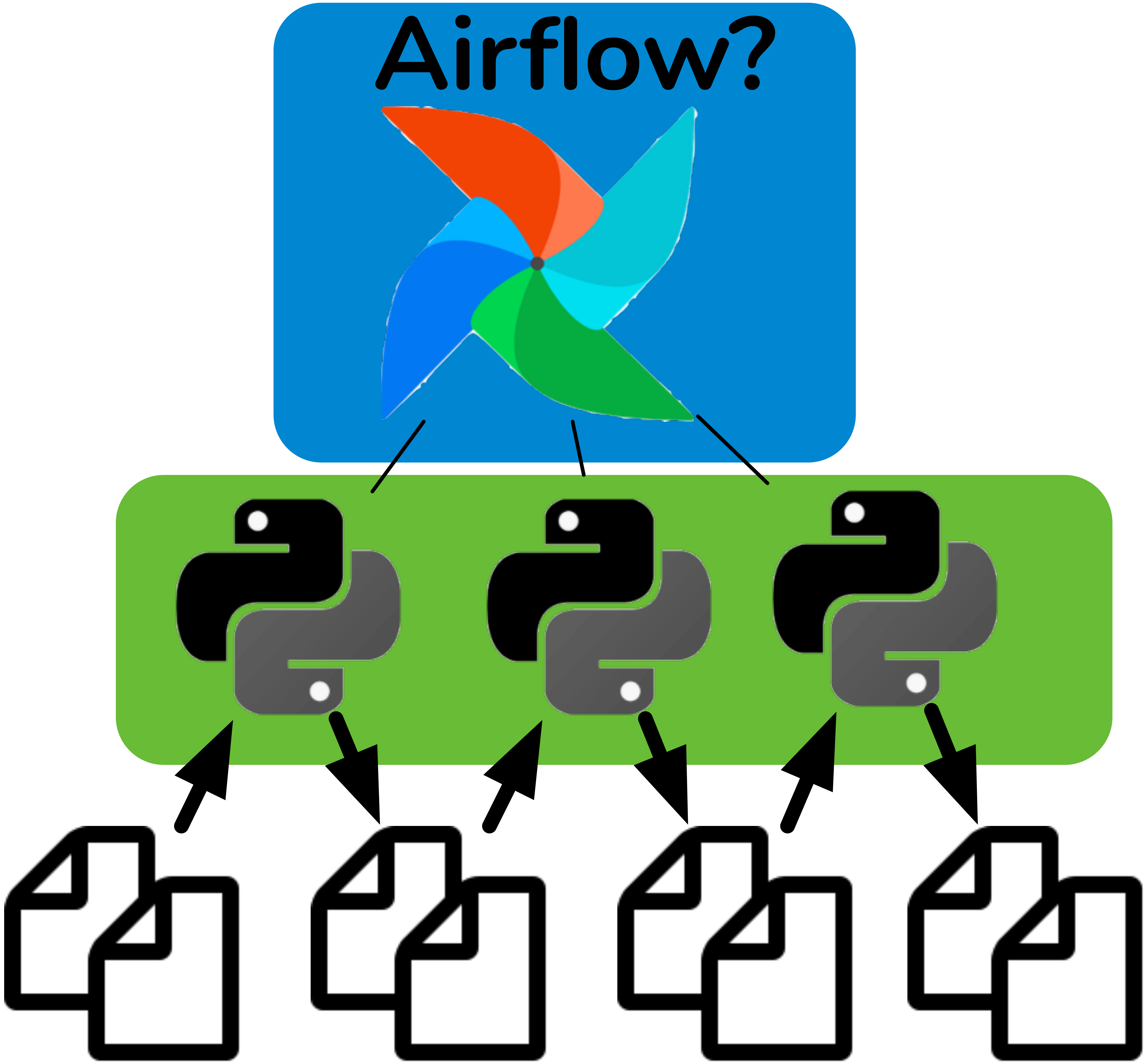


Warum dieser Vortrag (2)


Admin
Funktion

Endausbau

Warum dieser Vortrag (3)



Ziele des Vortrags

- > Konzepte
- > Idee, ob es sich lohnt
- > Wissen für die erste Woche
- > Airflow für Python
- > Cron ++++

~~>Profiwissen in Apache Airflow~~

~~>Betrieb von Airflow~~

Warum Workflows mit Apache Airflow?

Der Platz zwischen 'Enterprise' und 'Skript'

Skript	**Enterprise**
In Python, leicht zu erlernen und dem DS Team vertraut	Graphisch / Andere Sprache
Schnelle Implementierung	Strukturierte Ausführung, Logs im Lieferumfang
Rahmen muss Implementiert werden	Potentiell Lizenz- und Beratungskosten

Was ist Apache Airflow

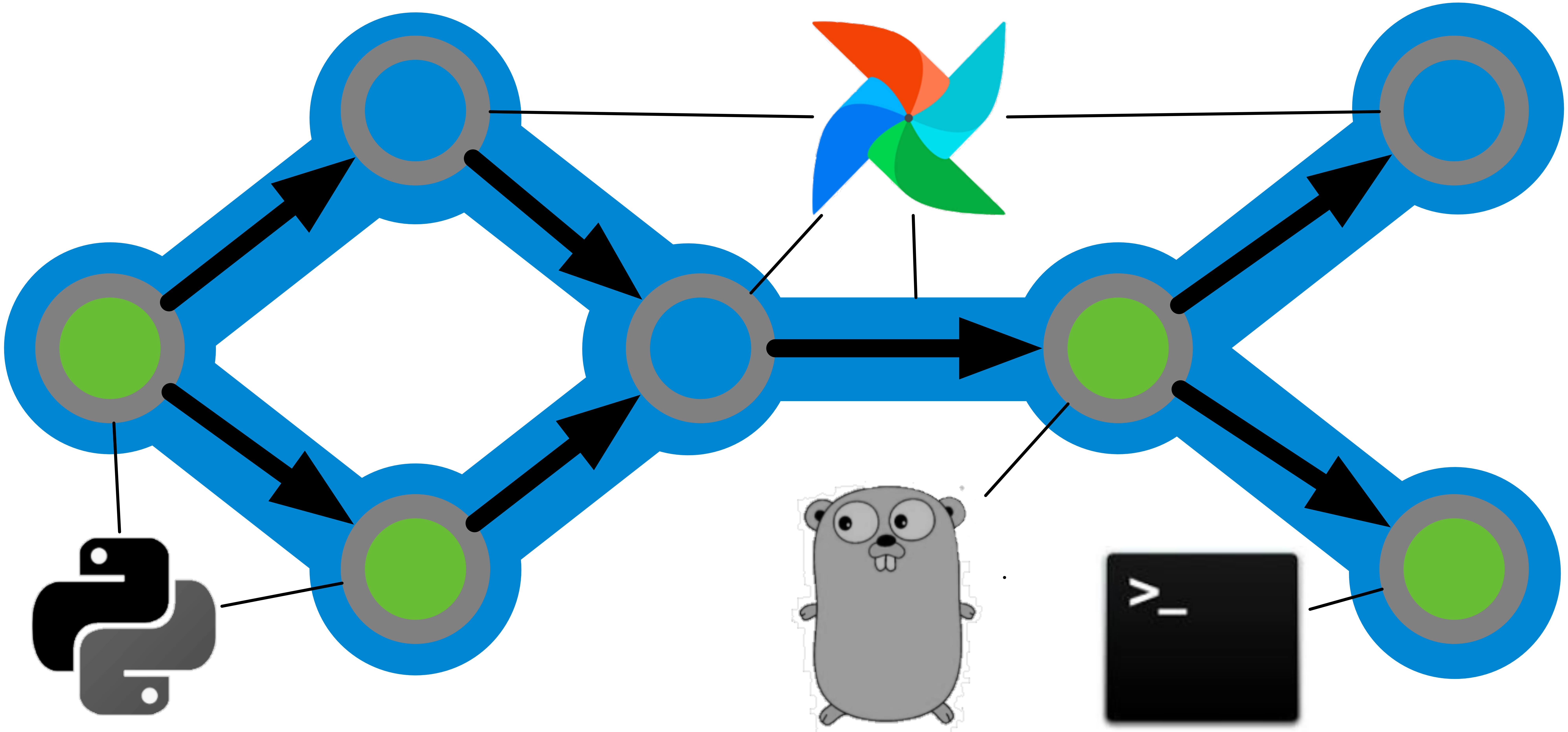
- Internes Airbnb Werkzeug seit 2014
- Seit 2016 Apache Incubator, seit 2019 Toplevel
- Werkzeug zur Definition und Ausführung von Batch Workflows
- Airflow ist in Python geschrieben und verwendet Python zur Definition von Workflows

Anforderungen und Airflow

	Data Science/DEV	IT Operations
Anforderung	> Flexible Implementierung > Geringer Aufwand für Integration	> Wartbarkeit > Monitorbarkeit > Skalierung / Nutzung von Infrastruktur
Airflow Funktionen	> Operatoren in beliebiger Sprache > Keine Annahme zu Datenspeichern	> Administration Oberfläche > Monitoring / Backfill > Clustering Support

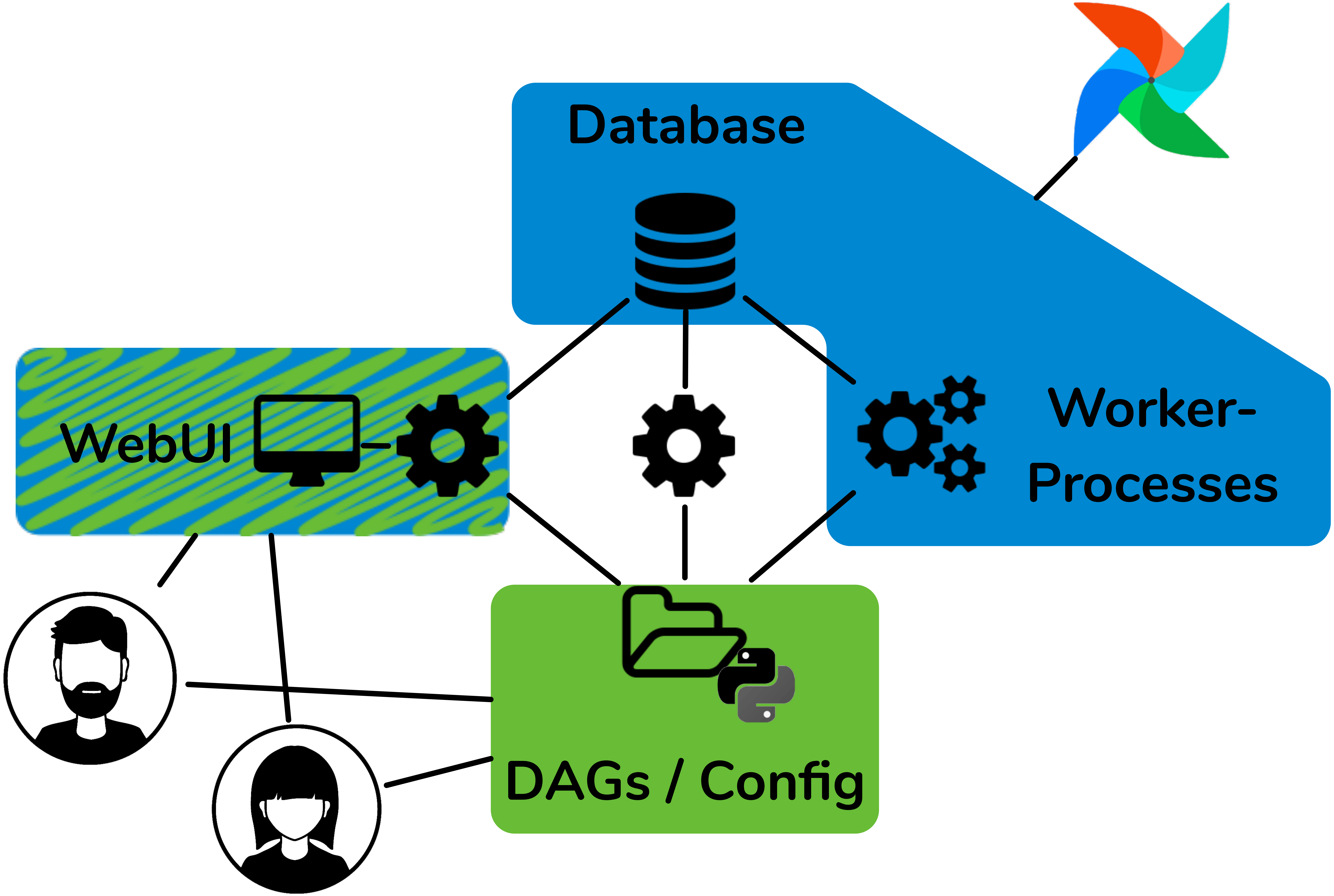
Airflow Konzepte

Airflow implementiert Workflows, Unterstützungsfunktionen und Standardoperatoren



Eigene Operatoren erlauben die entkoppelte Implementierung von Funktionalität

Architektur



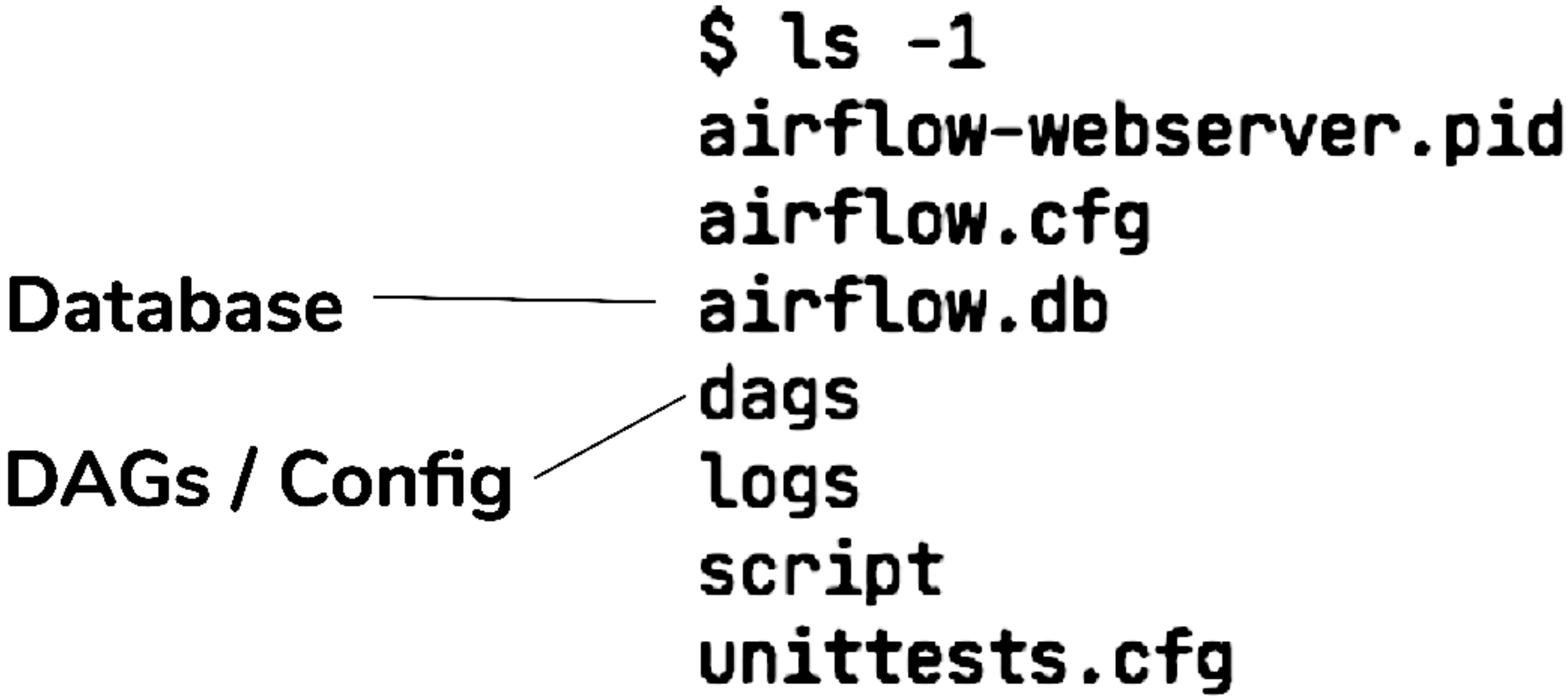
Database

WebUI

Worker-Processes

DAGs / Config

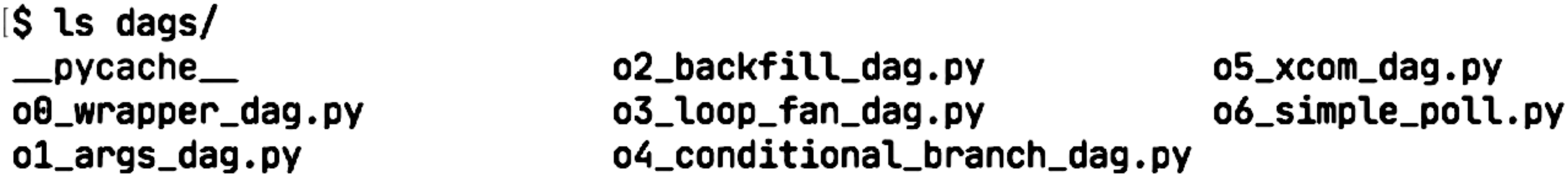
Verzeichnis und Dateien



```
$ ls -1
airflow-webserver.pid
airflow.cfg
airflow.db
dags
logs
script
unittests.cfg
```

Database

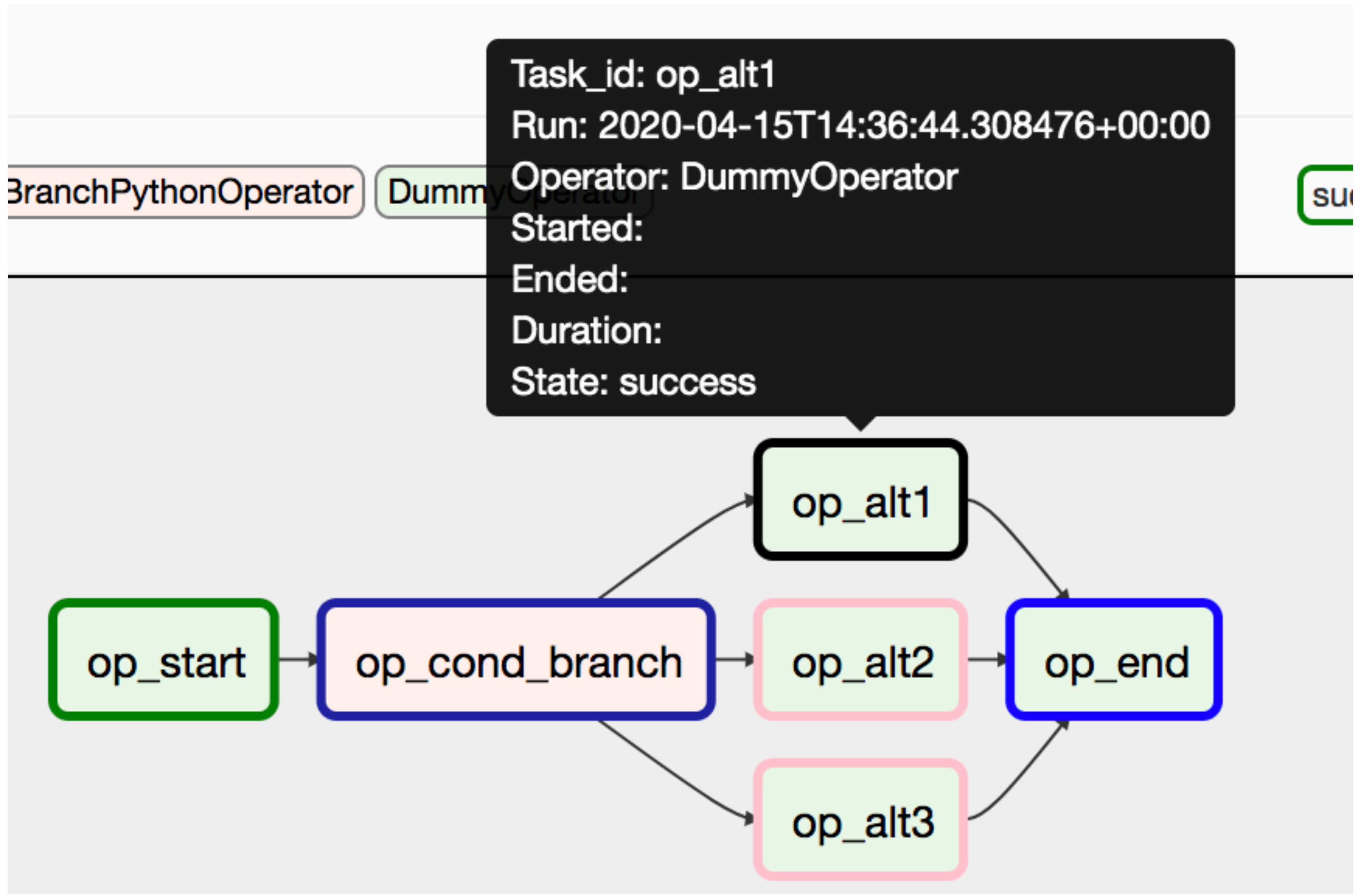
DAGs / Config



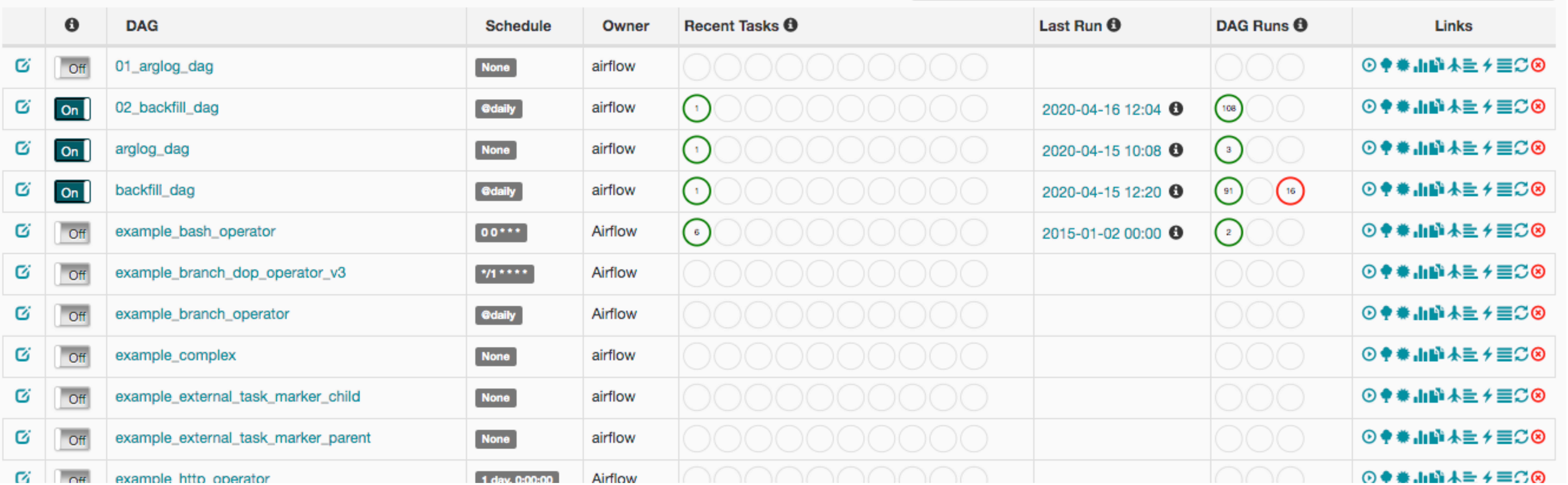
```
[$ ls dags/
 __pycache__                      o2_backfill_dag.py                o5_xcom_dag.py
 o0_wrapper_dag.py                o3_loop_fan_dag.py                o6_simple_poll.py
 o1_args_dag.py                   o4_conditional_branch_dag.py
```

Was bringt Airflow mit?

- Python API zur Definition von Workflows
- Scheduling, Ausführung und Monitoring von Tasks
- Benutzungsoberfläche und API
- Bibliothek von Schnittstellen und vorgefertigten Operationen



Task_id: op_alt1
Run: 2020-04-15T14:36:44.308476+00:00
Operator: DummyOperator
Started:
Ended:
Duration:
State: success



		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
	Off	01_arglog_dag	None	airflow				
	On	02_backfill_dag	@daily	airflow	1	2020-04-16 12:04	108	
	On	arglog_dag	None	airflow	1	2020-04-15 10:08	3	
	On	backfill_dag	@daily	airflow	1	2020-04-15 12:20	91 16	
	Off	example_bash_operator	0 0 * * *	Airflow	6	2015-01-02 00:00	2	
	Off	example_branch_dop_operator_v3	*/1 * * * *	Airflow				
	Off	example_branch_operator	@daily	Airflow				
	Off	example_complex	None	airflow				
	Off	example_external_task_marker_child	None	airflow				
	Off	example_external_task_marker_parent	None	airflow				
	Off	example_http_operator	1 day, 0:00:00	Airflow				

Scheduling, Trigger und Backfill

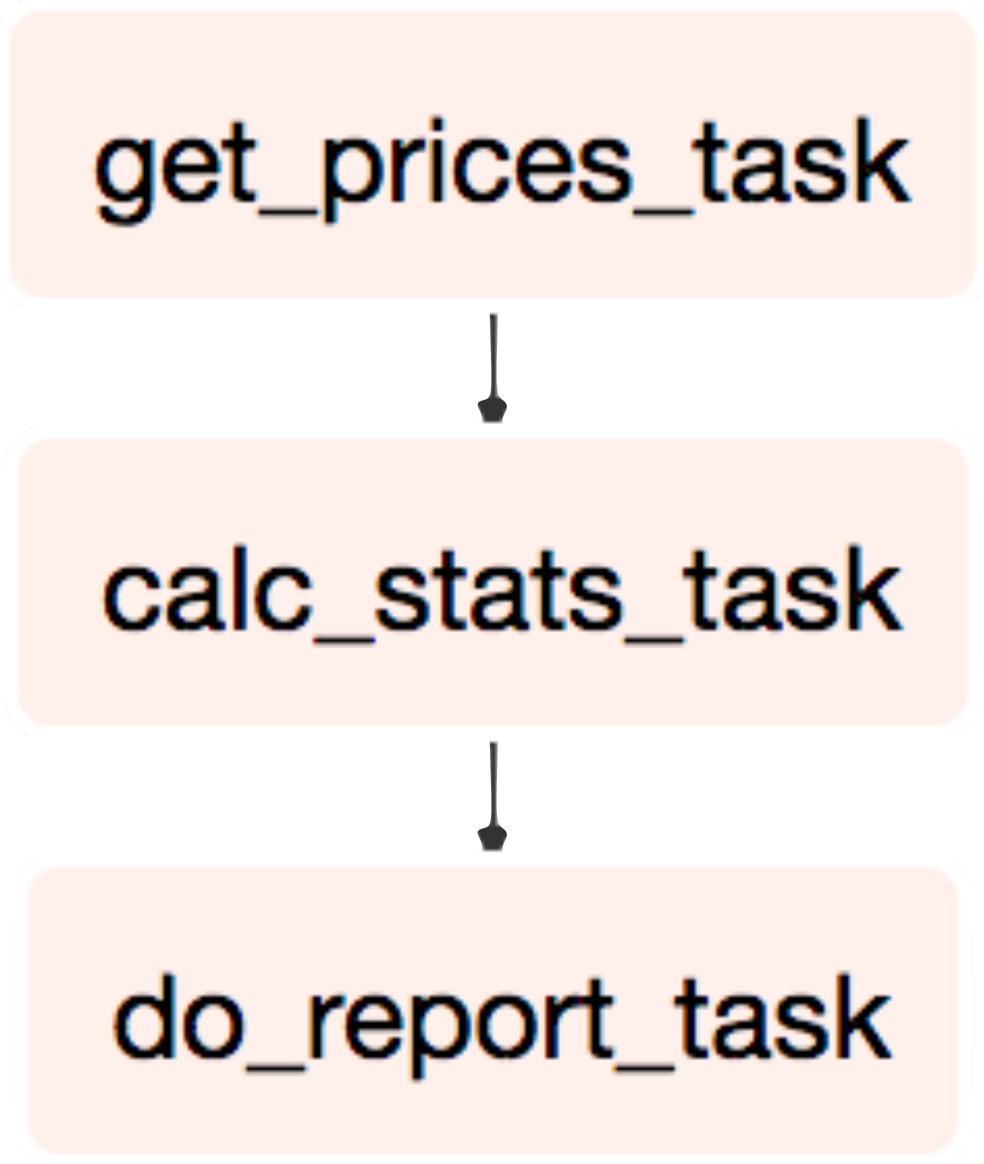
- Die Ausführung findet ausgehend vom <<start_date>> jeden <<schedule_interval>> statt.
- Möglichkeiten <<schedule_interval>> sind None, @once, @hourly, @daily, @weekly, @monthly, @yearly
- Fehlende Ausführungen können automatisch nachgeholt werden (backfill)

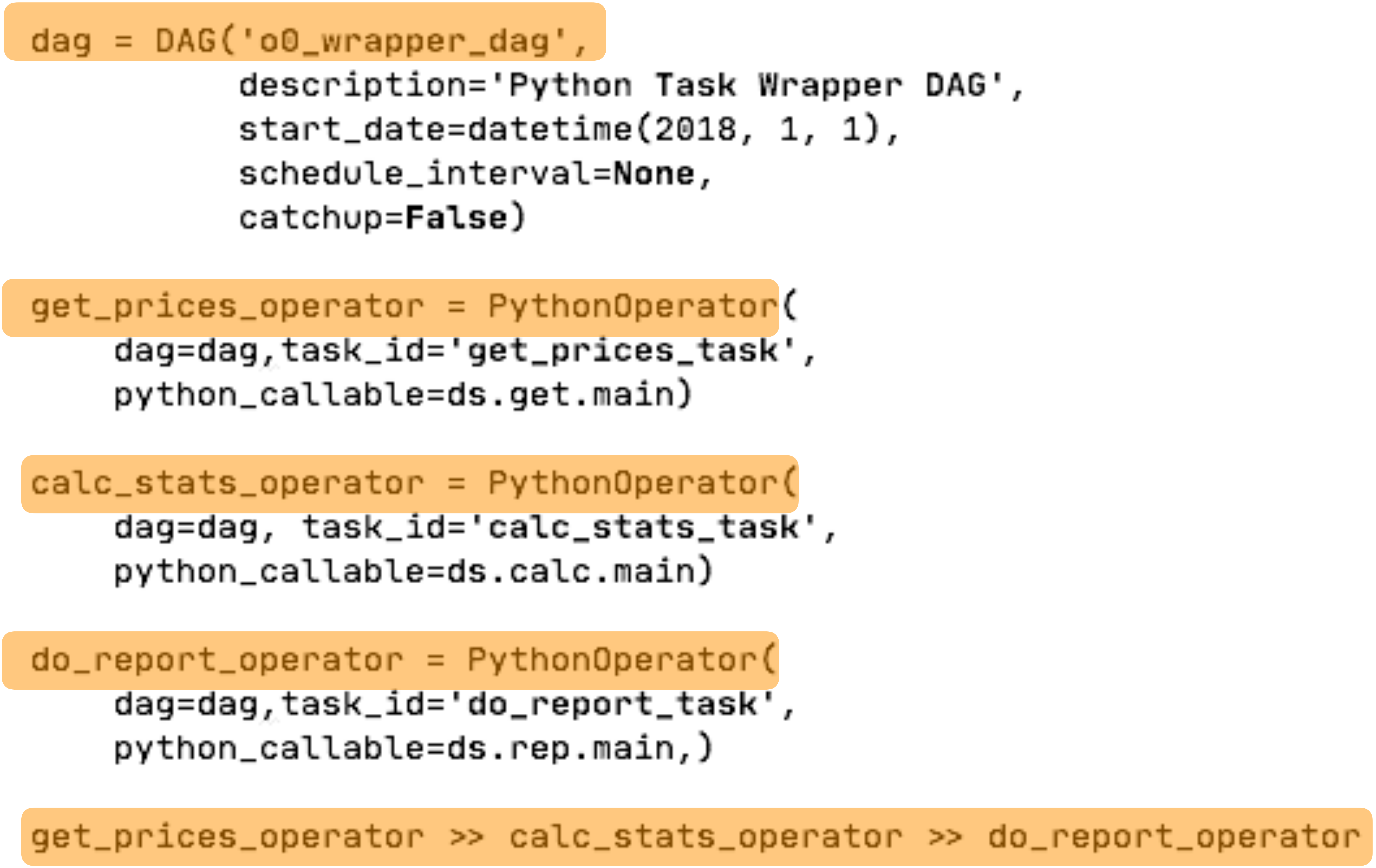
Kernkonzepte

DAG (=Directed, Acyclic Graph): Folge von Tasks

Task: Einzelner Workflow Schritt, implementiert durch Operator

Schedule: Zeitplan der Ausführung

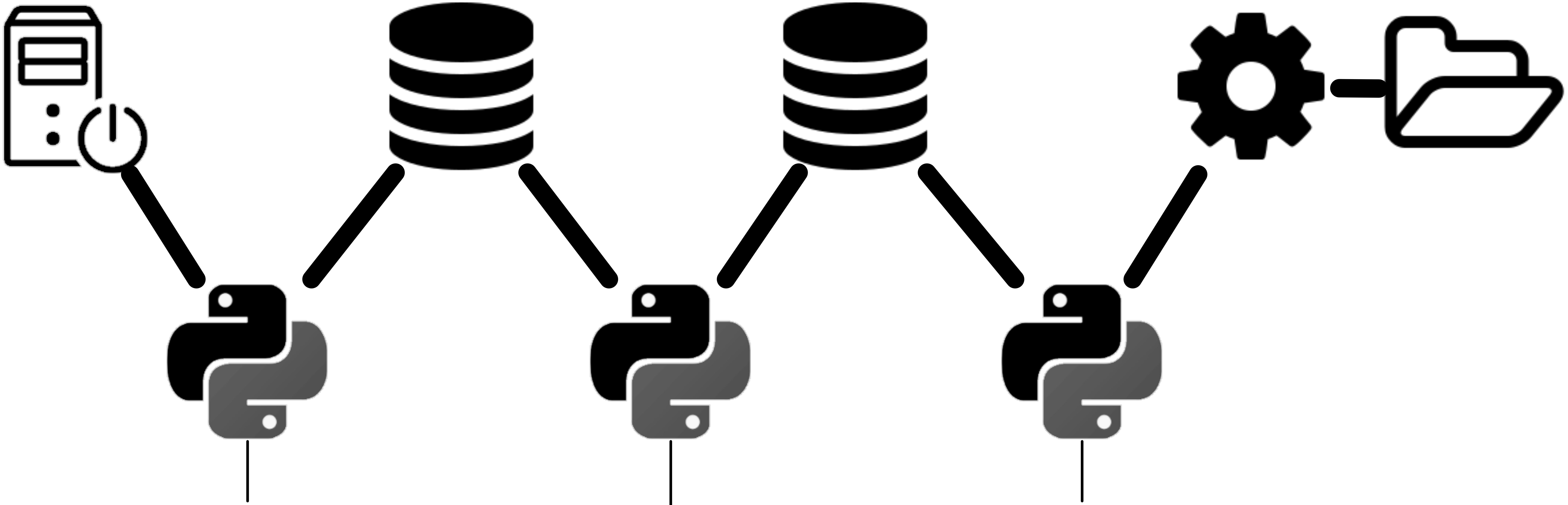




```
dag = DAG('o0_wrapper_dag',
          description='Python Task Wrapper DAG',
          start_date=datetime(2018, 1, 1),
          schedule_interval=None,
          catchup=False)

get_prices_operator = PythonOperator(
    dag=dag,task_id='get_prices_task',
    python_callable=ds.get.main)

calc_stats_operator = PythonOperator(
    dag=dag, task_id='calc_stats_task',
    python_callable=ds.calc.main)

do_report_operator = PythonOperator(
    dag=dag,task_id='do_report_task',
    python_callable=ds.rep.main,)

get_prices_operator >> calc_stats_operator >> do_report_operator
```

Überführung von Ad-Hoc Skripten nach Airflow

Das Szenario



Daten von API abrufen
Berechnungen vornehmen
In Datenbank schreiben

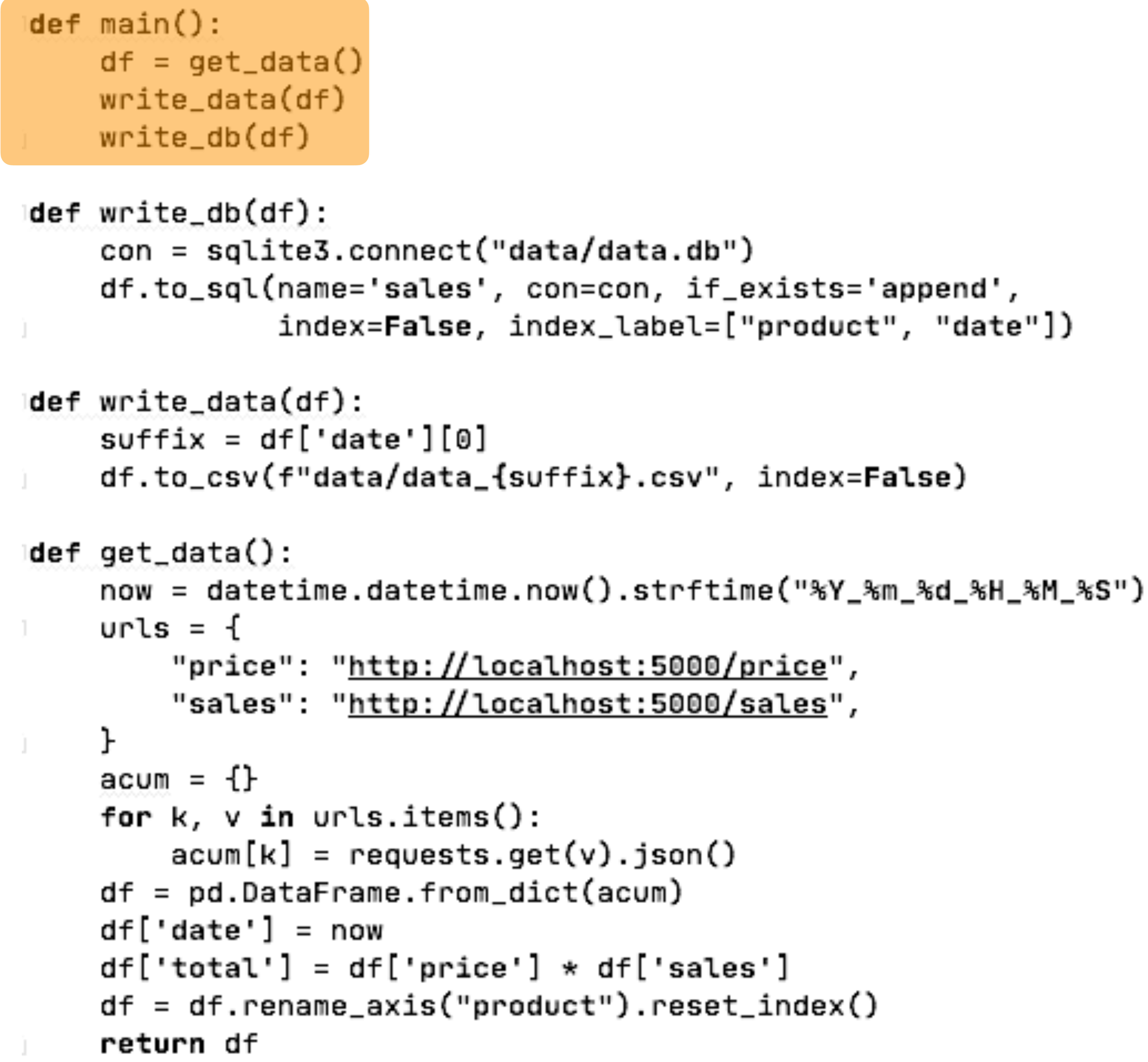
Aus Datenbank lesen
PPTX Report erstellen
PPTX auf FTP Server schreiben

Aus Datenbank lesen
Daten akkumulieren
Daten in Datenbank schreiben

https://github.com/stbaercom/enterpy_airflow

Beispielskripte (1/3)

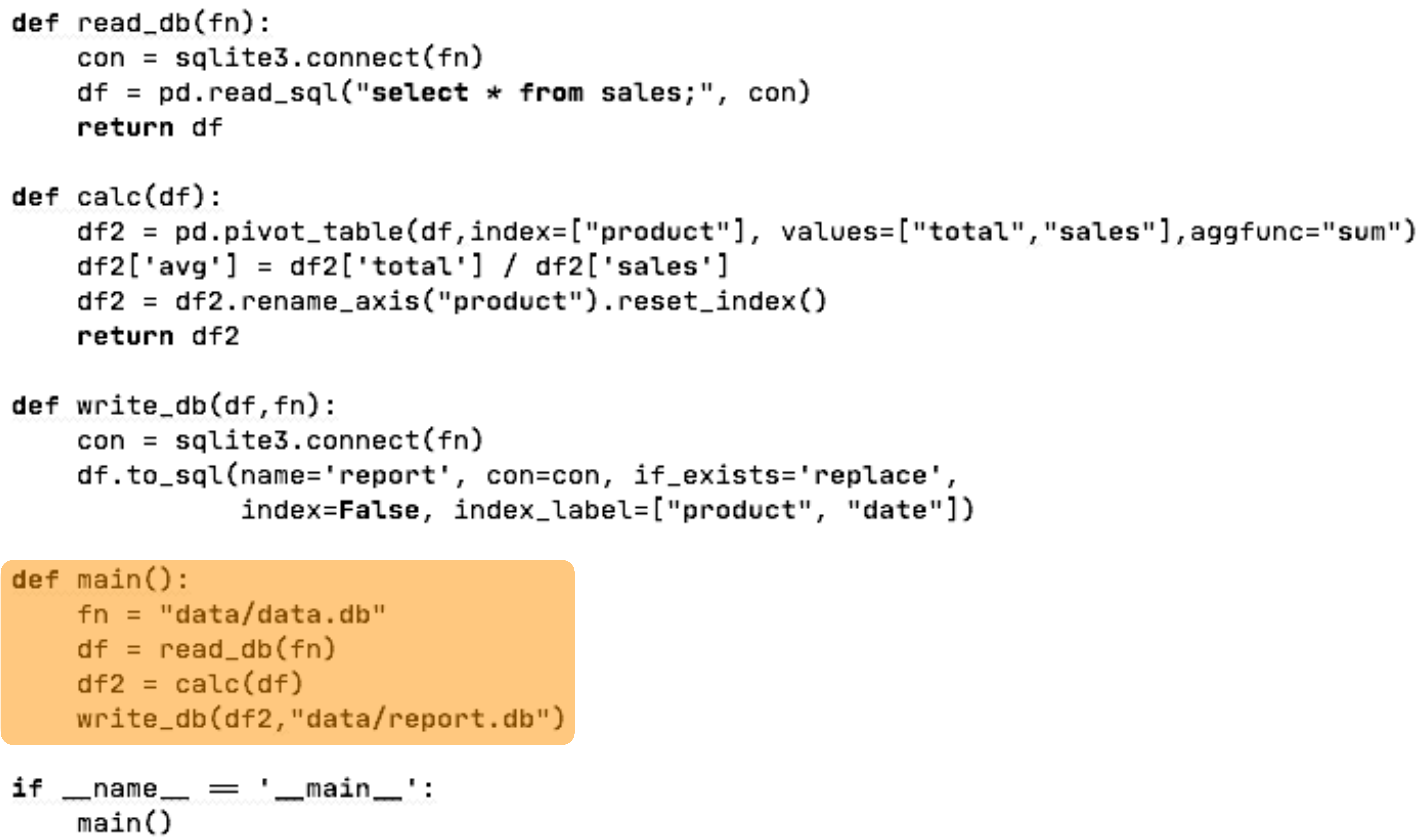
- Schritt 1 - API Abruf / Berechnung / Datenbank und CSV



```python
def main():
    df = get_data()
    write_data(df)
    write_db(df)

def write_db(df):
    con = sqlite3.connect("data/data.db")
    df.to_sql(name='sales', con=con, if_exists='append',
              index=False, index_label=["product", "date"])

def write_data(df):
    suffix = df['date'][0]
    df.to_csv(f"data/data_{suffix}.csv", index=False)

def get_data():
    now = datetime.datetime.now().strftime("%Y_%m_%d_%H_%M_%S")
    urls = {
        "price": "http://localhost:5000/price",
        "sales": "http://localhost:5000/sales",
    }
    acum = {}
    for k, v in urls.items():
        acum[k] = requests.get(v).json()
    df = pd.DataFrame.from_dict(acum)
    df['date'] = now
    df['total'] = df['price'] * df['sales']
    df = df.rename_axis("product").reset_index()
    return df
```

Beispielskripte (2/3)

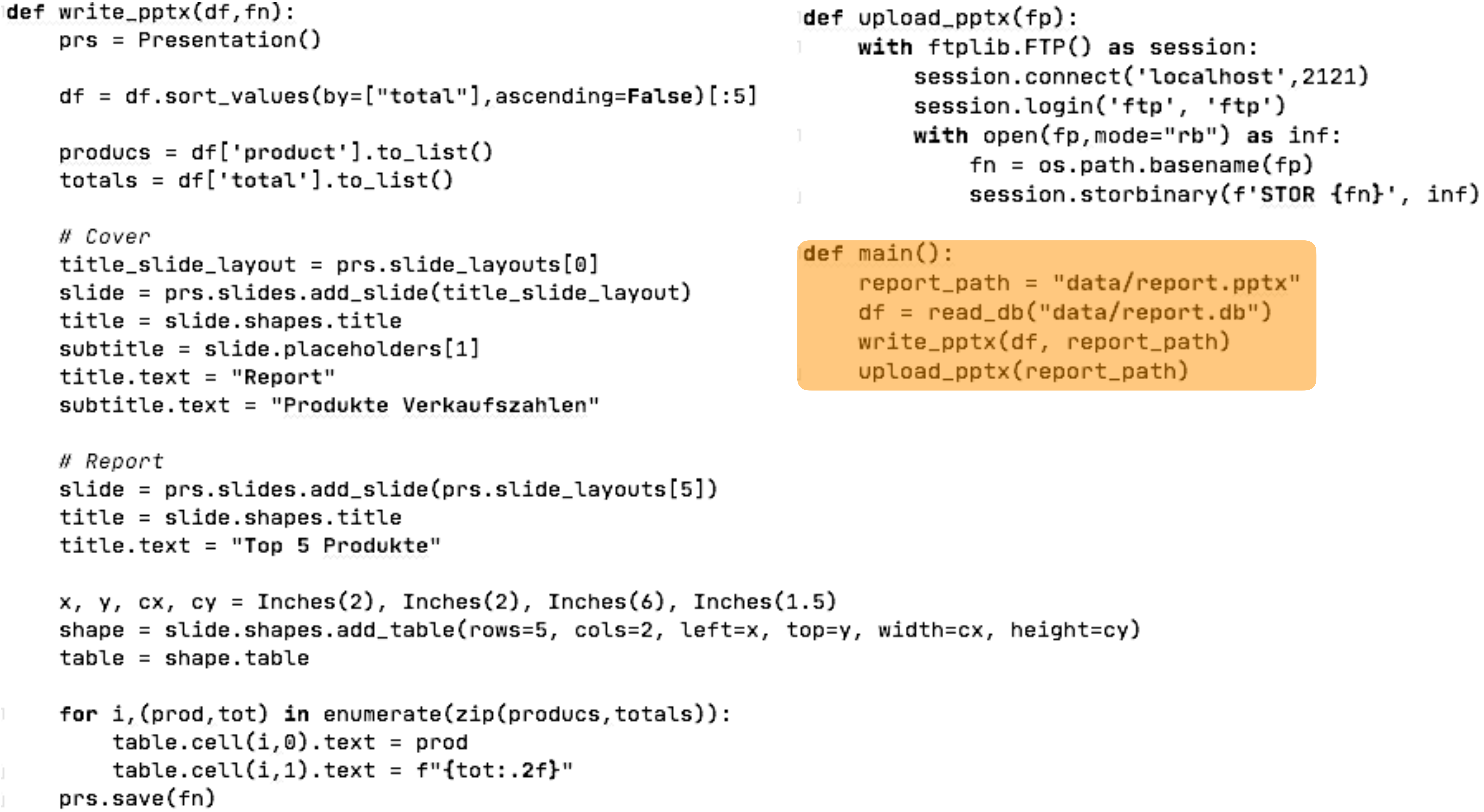
- Schritt 2 - Lesen Datenbank / Berechnung / Zurückschreiben



```
def read_db(fn):
    con = sqlite3.connect(fn)
    df = pd.read_sql("select * from sales;", con)
    return df

def calc(df):
    df2 = pd.pivot_table(df,index=["product"], values=["total","sales"],aggfunc="sum")
    df2['avg'] = df2['total'] / df2['sales']
    df2 = df2.rename_axis("product").reset_index()
    return df2

def write_db(df,fn):
    con = sqlite3.connect(fn)
    df.to_sql(name='report', con=con, if_exists='replace',
              index=False, index_label=["product", "date"])

def main():
    fn = "data/data.db"
    df = read_db(fn)
    df2 = calc(df)
    write_db(df2,"data/report.db")

if __name__ == '__main__':
    main()
```

Beispielskripte (3/3)

- Schritt 3 - PPTX Präsentation / Auf FTP Server schreiben

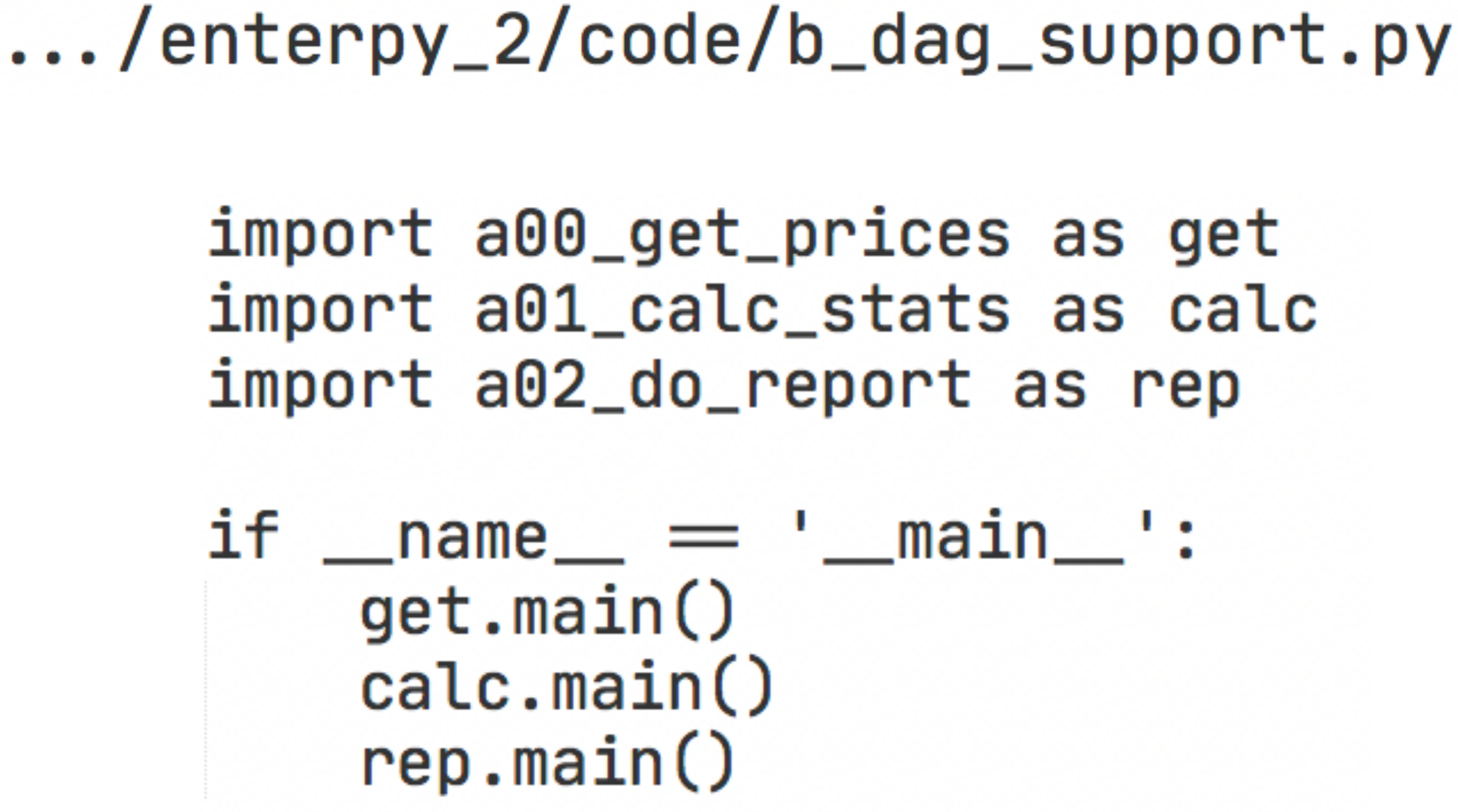


```
def write_pptx(df,fn):
    prs = Presentation()

    df = df.sort_values(by=["total"],ascending=False)[:5]

    producs = df['product'].to_list()
    totals = df['total'].to_list()

    # Cover
    title_slide_layout = prs.slide_layouts[0]
    slide = prs.slides.add_slide(title_slide_layout)
    title = slide.shapes.title
    subtitle = slide.placeholders[1]
    title.text = "Report"
    subtitle.text = "Produkte Verkaufszahlen"

    # Report
    slide = prs.slides.add_slide(prs.slide_layouts[5])
    title = slide.shapes.title
    title.text = "Top 5 Produkte"

    x, y, cx, cy = Inches(2), Inches(2), Inches(6), Inches(1.5)
    shape = slide.shapes.add_table(rows=5, cols=2, left=x, top=y, width=cx, height=cy)
    table = shape.table

    for i,(prod,tot) in enumerate(zip(producs,totals)):
        table.cell(i,0).text = prod
        table.cell(i,1).text = f"{tot:.2f}"
    prs.save(fn)
```

```
def upload_pptx(fp):
    with ftplib.FTP() as session:
        session.connect('localhost',2121)
        session.login('ftp', 'ftp')
        with open(fp,mode="rb") as inf:
            fn = os.path.basename(fp)
            session.storbinary(f'STOR {fn}', inf)
```

```
def main():
    report_path = "data/report.pptx"
    df = read_db("data/report.db")
    write_pptx(df, report_path)
    upload_pptx(report_path)
```

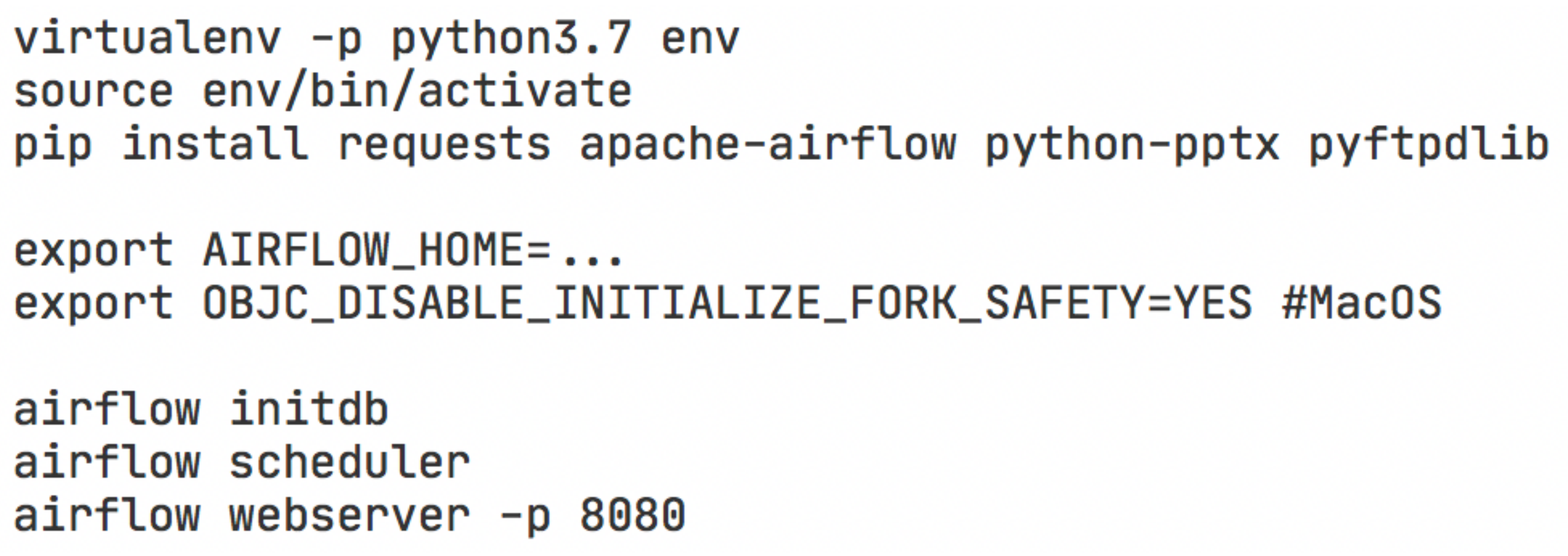
Skript ohne Airflow



```
.../enterpy_2/code/b_dag_support.py
```

```
import a00_get_prices as get
import a01_calc_stats as calc
import a02_do_report as rep

if __name__ == '__main__':
    get.main()
    calc.main()
    rep.main()
```

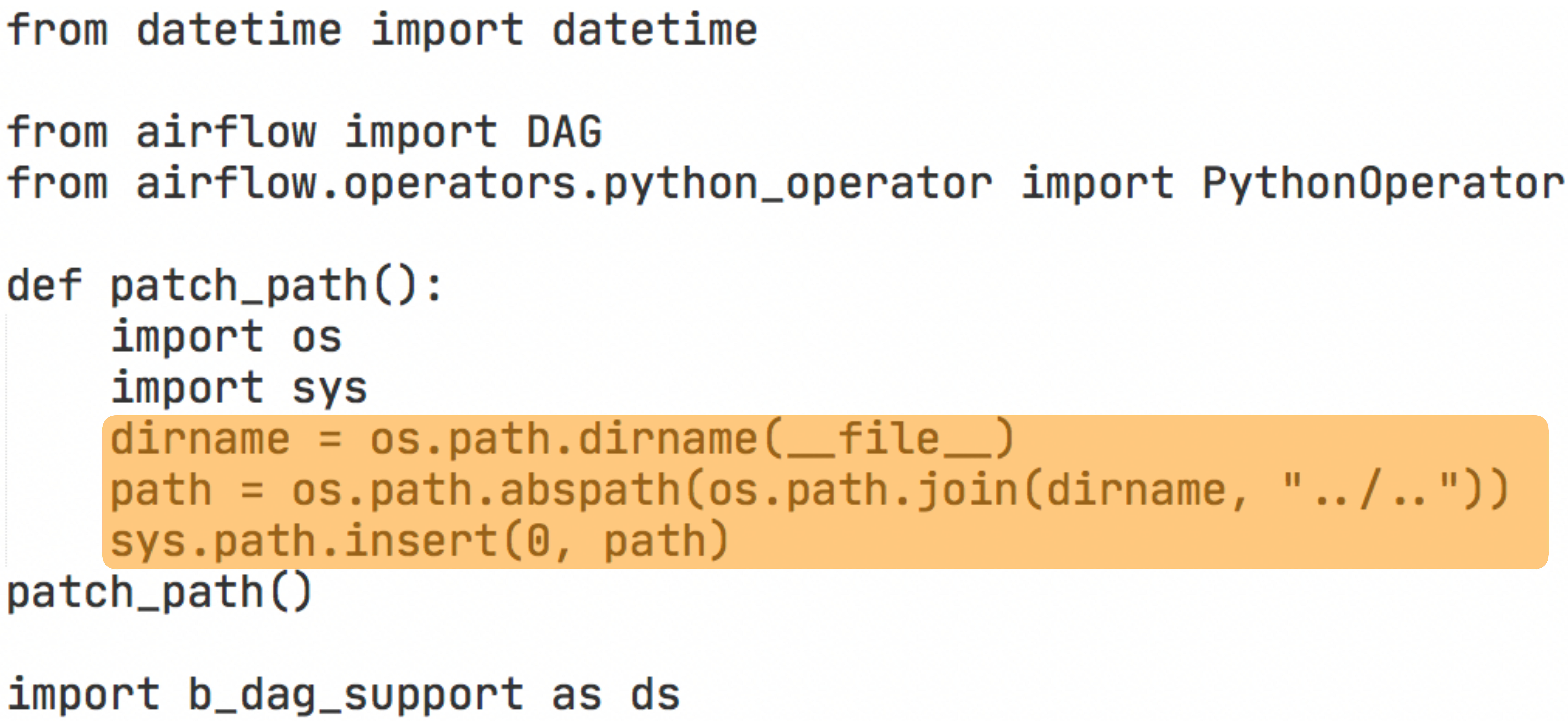
Apache Airflow Installieren und Starten



```
virtualenv -p python3.7 env
source env/bin/activate
pip install requests apache-airflow python-pptx pyftpdlib

export AIRFLOW_HOME=...
export OBJC_DISABLE_INITIALIZE_FORK_SAFETY=YES #MacOS

airflow initdb
airflow scheduler
airflow webserver -p 8080
```

Direkte Portierung nach Airflow

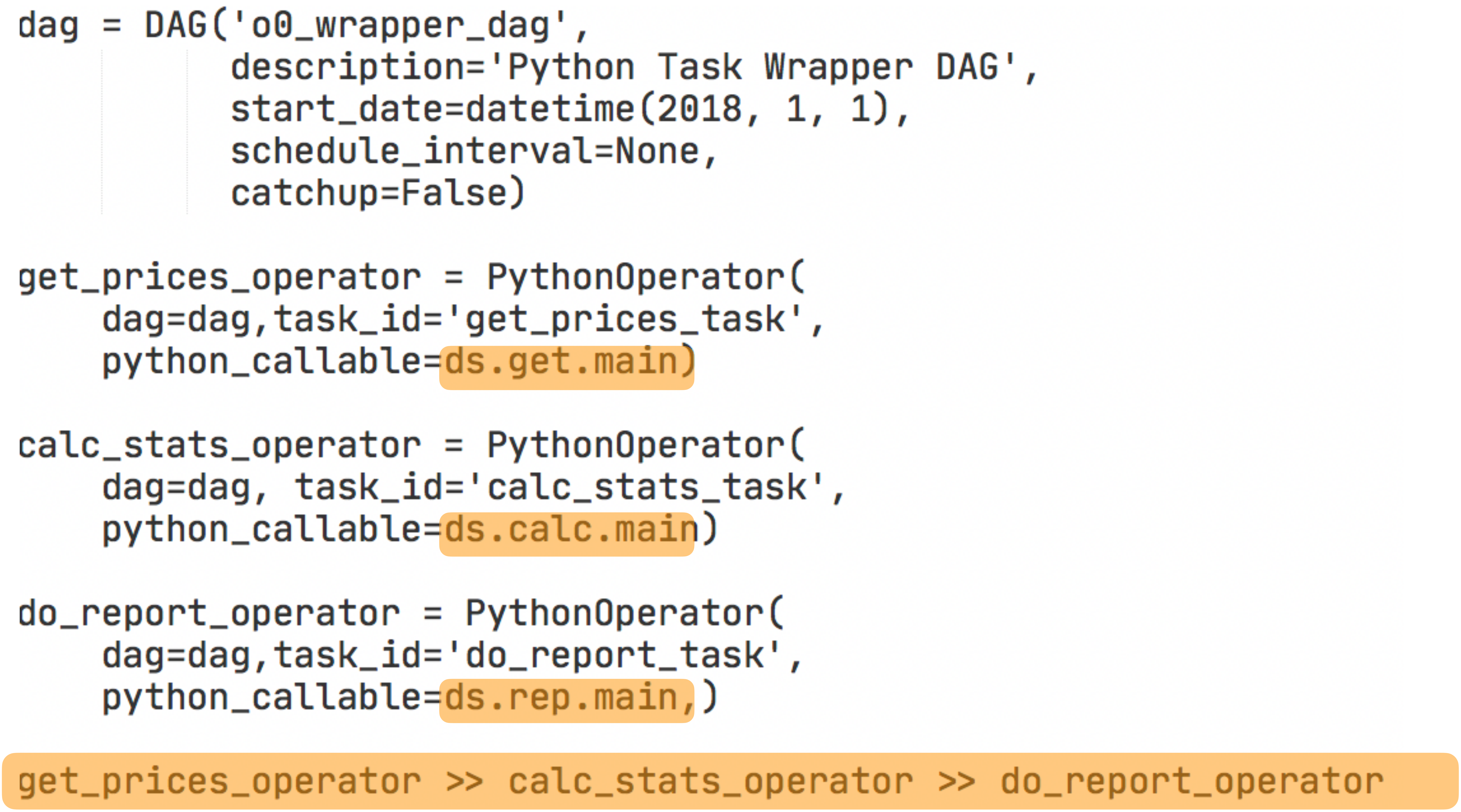
`.../enterpy_2/code/airflow_home/dags/o0_wrapper_dag.py`



```
from datetime import datetime

from airflow import DAG
from airflow.operators.python_operator import PythonOperator

def patch_path():
    import os
    import sys
    dirname = os.path.dirname(__file__)
    path = os.path.abspath(os.path.join(dirname, "../.."))
    sys.path.insert(0, path)
patch_path()

import b_dag_support as ds
```

Direkte Portierung nach Airflow

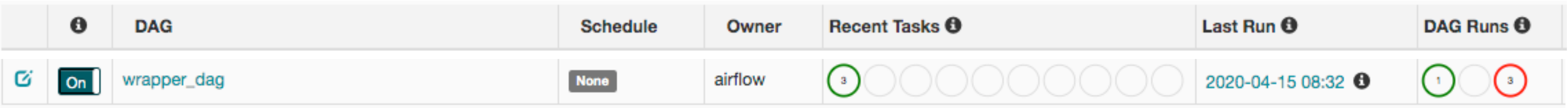
`.../enterpy_2/code/airflow_home/dags/o0_wrapper_dag.py`



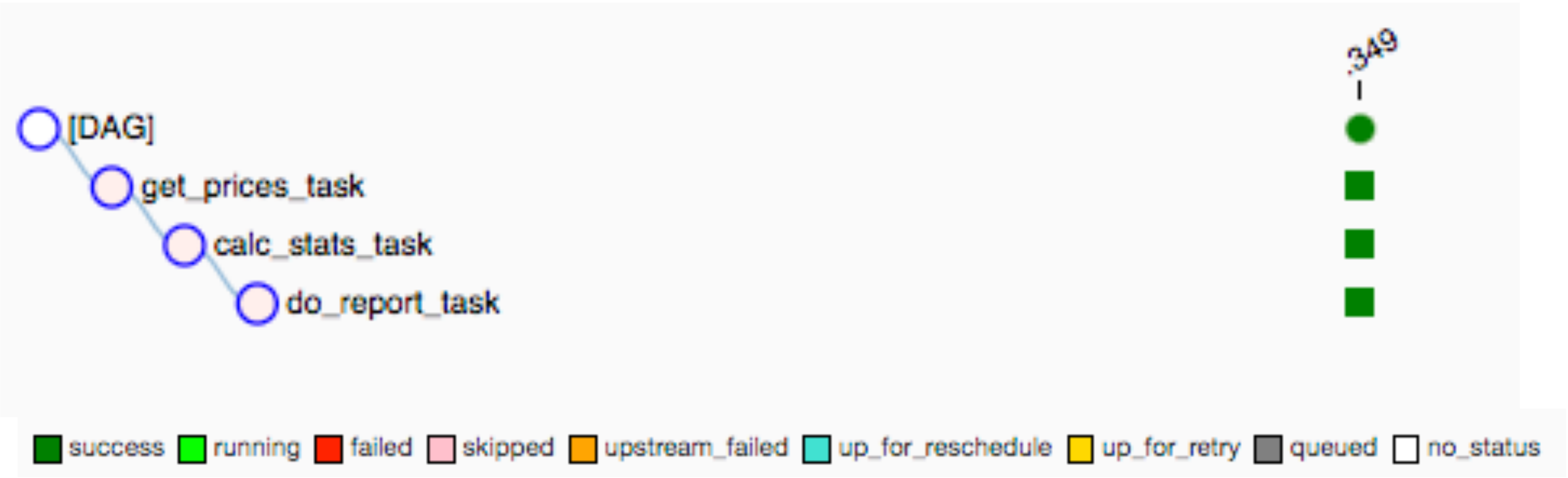
```
dag = DAG('o0_wrapper_dag',
          description='Python Task Wrapper DAG',
          start_date=datetime(2018, 1, 1),
          schedule_interval=None,
          catchup=False)

get_prices_operator = PythonOperator(
    dag=dag,task_id='get_prices_task',
    python_callable=ds.get.main)

calc_stats_operator = PythonOperator(
    dag=dag, task_id='calc_stats_task',
    python_callable=ds.calc.main)

do_report_operator = PythonOperator(
    dag=dag,task_id='do_report_task',
    python_callable=ds.rep.main,)

get_prices_operator >> calc_stats_operator >> do_report_operator
```

Vorteile der direkten Portierung

- Keine Anpassung am Quellcode notwendig
- Scheduling und automatische Ausführung über Airflow
- Dokumentation der erfolgten Ausführung
- Möglichkeit der teilweisen Ausführung



	ℹ	DAG	Schedule	Owner	Recent Tasks ℹ	Last Run ℹ	DAG Runs ℹ
	On	wrapper_dag	None	airflow	3	2020-04-15 08:32 ℹ	1 3


.349
[DAG]
get_prices_task
calc_stats_task
do_report_task
success
running
failed
skipped
upstream_failed
up_for_reschedule
up_for_retry
queued
no_status

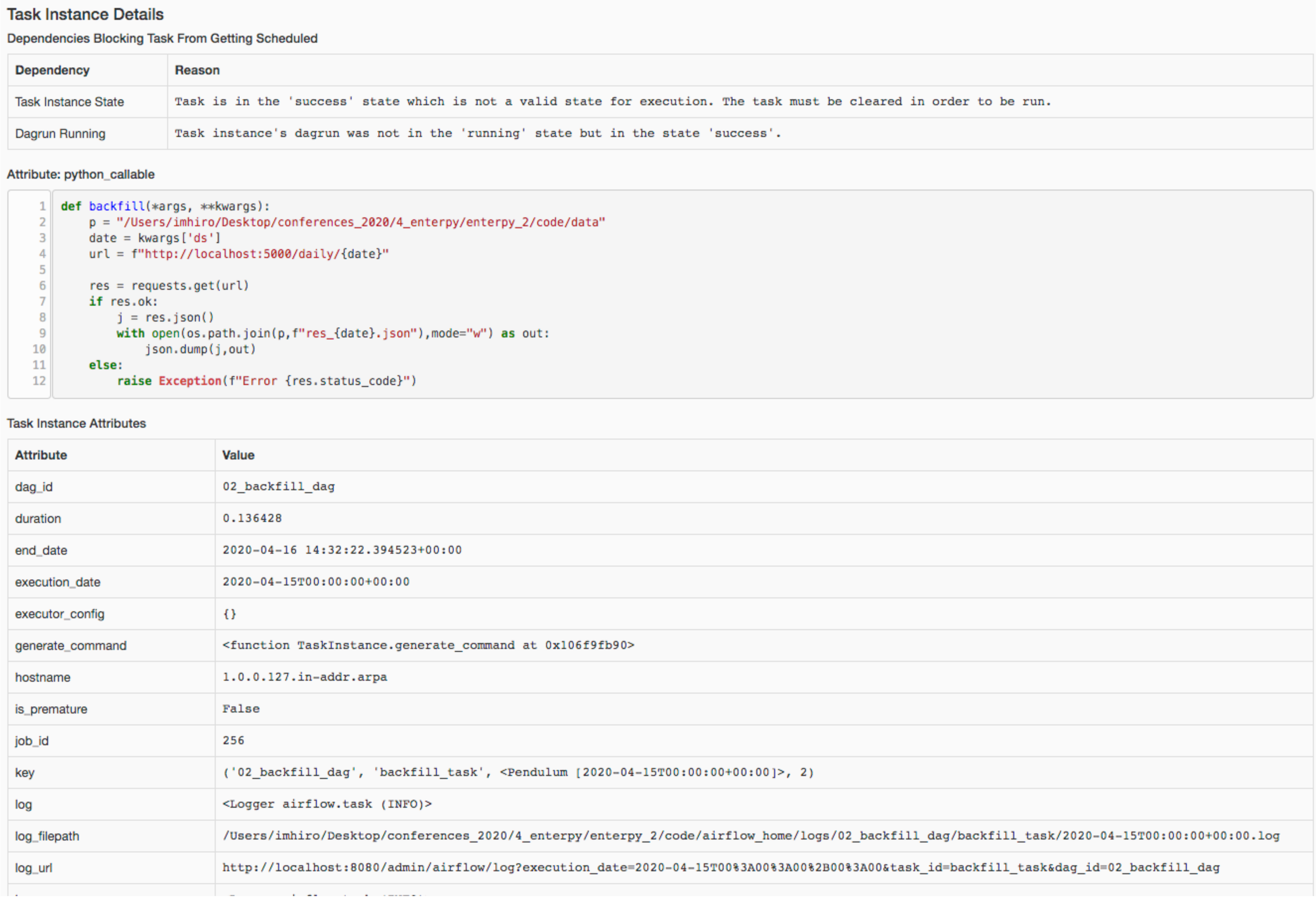
Übersicht zu Tasks

List (257)
Add Filter
With selected
Search: dag_id, task_id, state
State
Dag Id
Task Id
Execution Date
Hostname
Queue
Pool
Operator
Start Date
End Date

Task Instances

☐		State		Task Id	Execution Date	Operator	Start Date	End Date	Duration	Job Id	Hostname	Unixname	P V
☐		succ		backfill_task	04-16T00:00:00+00:00	PythonOperator	04-17T00:00:11.562633+00:00	04-17T00:00:11.703637+00:00	0:00:00.141004	257	1.0.0.127.in-addr.arpa	imhiro	1
☐		succ		backfill_task	04-15T00:00:00+00:00	PythonOperator	04-16T14:32:22.258095+00:00	04-16T14:32:22.394523+00:00	0:00:00.136428	256	1.0.0.127.in-addr.arpa	imhiro	1
☐		succ		backfill_task	04-14T00:00:00+00:00	PythonOperator	04-16T14:31:08.528141+00:00	04-16T14:31:08.690648+00:00	0:00:00.162507	255	1.0.0.127.in-addr.arpa	imhiro	1
☐		succ		backfill_task	04-13T00:00:00+00:00	PythonOperator	04-16T14:29:53.892747+00:00	04-16T14:29:54.087605+00:00	0:00:00.194858	254	1.0.0.127.in-addr.arpa	imhiro	1
☐		succ		backfill_task	04-12T00:00:00+00:00	PythonOperator	04-16T14:28:40.402310+00:00	04-16T14:28:40.528839+00:00	0:00:00.126529	253	1.0.0.127.in-addr.arpa	imhiro	1
☐		succ		backfill_task	04-11T00:00:00+00:00	PythonOperator	04-16T14:27:27.264846+00:00	04-16T14:27:27.382023+00:00	0:00:00.117177	252	1.0.0.127.in-addr.arpa	imhiro	1
☐		success	02_backfill_dag	backfill_task	04-10T00:00:00+00:00	PythonOperator	04-16T14:26:14.083898+00:00	04-16T14:26:14.225918+00:00	0:00:00.142020	251	1.0.0.127.in-addr.arpa	imhiro	1
☐		success	02_backfill_dag	backfill_task	04-09T00:00:00+00:00	PythonOperator	04-16T14:25:01.383505+00:00	04-16T14:25:01.508360+00:00	0:00:00.124855	250	1.0.0.127.in-addr.arpa	imhiro	1
☐		success	02_backfill_dag	backfill_task	04-08T00:00:00+00:00	PythonOperator	04-16T14:23:48.728294+00:00	04-16T14:23:48.853335+00:00	0:00:00.125041	249	1.0.0.127.in-addr.arpa	imhiro	1
☐		success	02_backfill_dag	backfill_task	04-07T00:00:00+00:00	PythonOperator	04-16T14:22:35.792017+00:00	04-16T14:22:35.941269+00:00	0:00:00.149252	248	1.0.0.127.in-addr.arpa	imhiro	1
☐		success	02_backfill_dag	backfill_task	04-06T00:00:00+00:00	PythonOperator	04-16T14:21:23.160619+00:00	04-16T14:21:23.289857+00:00	0:00:00.129238	247	1.0.0.127.in-addr.arpa	imhiro	1

Task Details



Task Instance Details

Dependencies Blocking Task From Getting Scheduled

Dependency	Reason
Task Instance State	`Task is in the 'success' state which is not a valid state for execution. The task must be cleared in order to be run.`
Dagrun Running	`Task instance's dagrun was not in the 'running' state but in the state 'success'.`

Attribute: python_callable

```python
def backfill(*args, **kwargs):
    p = "/Users/imhiro/Desktop/conferences_2020/4_enterpy/enterpy_2/code/data"
    date = kwargs['ds']
    url = f"http://localhost:5000/daily/{date}"

    res = requests.get(url)
    if res.ok:
        j = res.json()
        with open(os.path.join(p,f"res_{date}.json"),mode="w") as out:
            json.dump(j,out)
    else:
        raise Exception(f"Error {res.status_code}")
```

Task Instance Attributes

Attribute	Value
dag_id	02_backfill_dag
duration	0.136428
end_date	2020-04-16 14:32:22.394523+00:00
execution_date	2020-04-15T00:00:00+00:00
executor_config	{}
generate_command	<function TaskInstance.generate_command at 0x106f9fb90>
hostname	1.0.0.127.in-addr.arpa
is_premature	False
job_id	256
key	('02_backfill_dag', 'backfill_task', <Pendulum [2020-04-15T00:00:00+00:00]>, 2)
log	<Logger airflow.task (INFO)>
log_filepath	/Users/imhiro/Desktop/conferences_2020/4_enterpy/enterpy_2/code/airflow_home/logs/02_backfill_dag/backfill_task/2020-04-15T00:00:00+00:00.log
log_url	http://localhost:8080/admin/airflow/log?execution_date=2020-04-15T00%3A00%3A00%2B00%3A00&task_id=backfill_task&dag_id=02_backfill_dag

Nächste Schritte

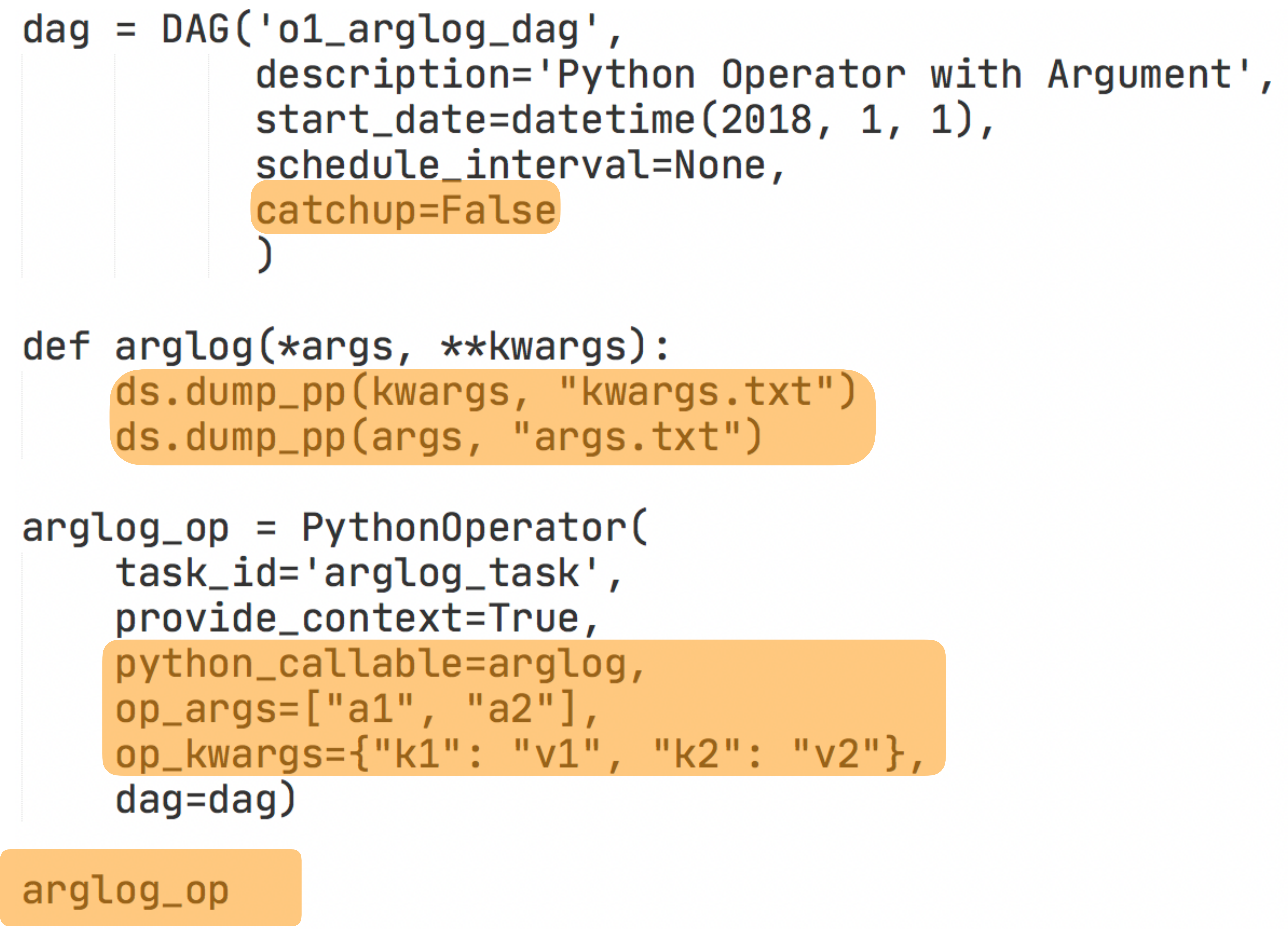
- Die aktuelle Implementierung basiert 1-zu-1 auf den bestehenden Skripten
- Für die weitere Entwicklung stehen zahlreiche Möglichkeiten zur Verfügung
  - Aufteilung der Skripte in Teilschritte
  - Verwendungen von passenden Airflow Operatoren, um weniger eigenen Code zu verwenden (besonders nützlich, wenn auf Clouddienste / Datenbanken gewechselt werden soll)
- Das Deployment sollte noch angepasst werden. Bei Verwendung von SQLite als Datenbank ist Airflow auf eine Ausführung zur Zeit beschränkt.

Airflow Beispiele

Ziele der Beispiele

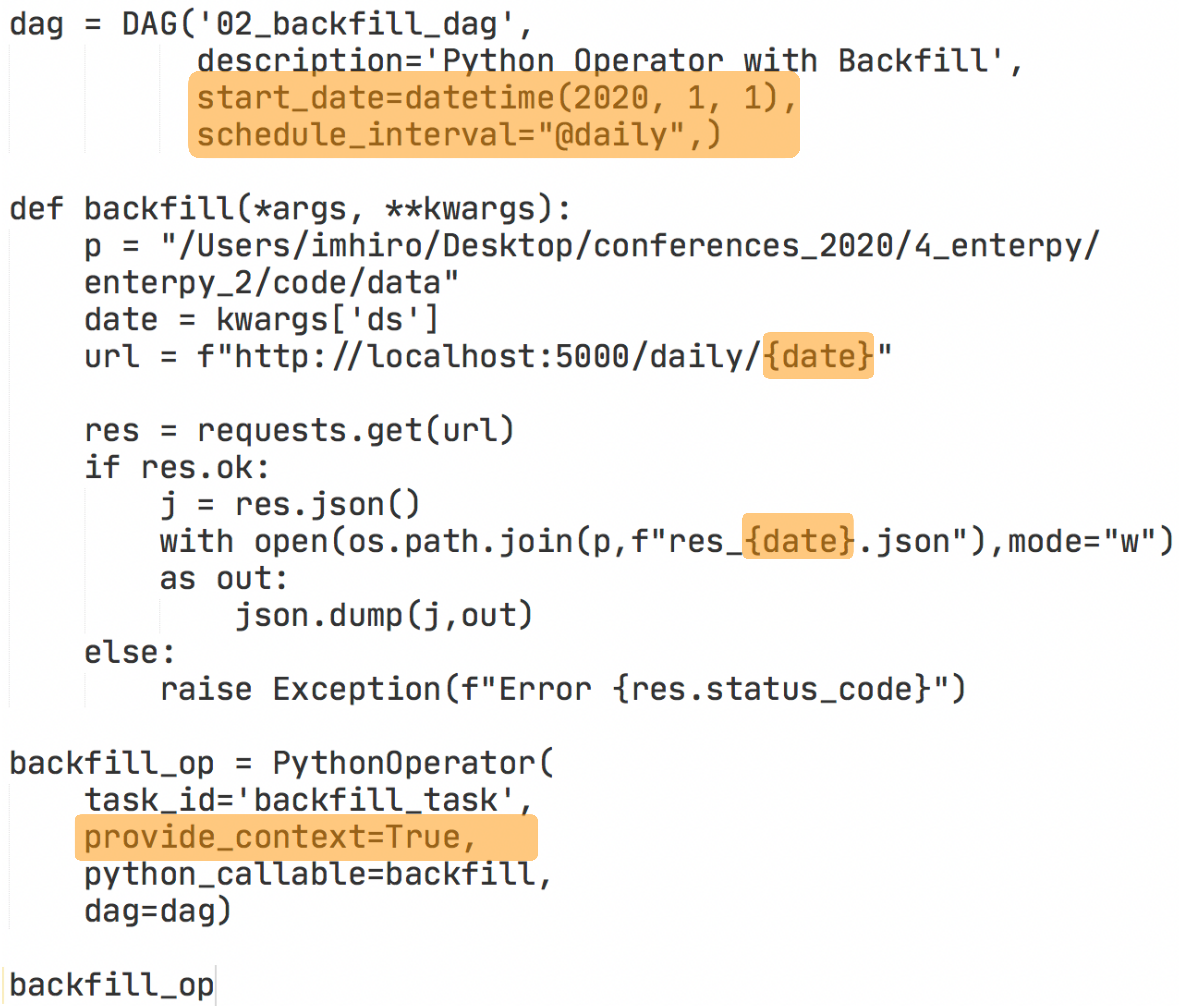
- Umsetzung von Standardaufgaben mit möglichst einfachen Mitteln
  - Keine Implementierung von eigenen Operatoren, Signalen, etc.
- Darstellung der 60% Funktionalität für 80% aller Aufgaben.

Argumente und Kontext für Skripte



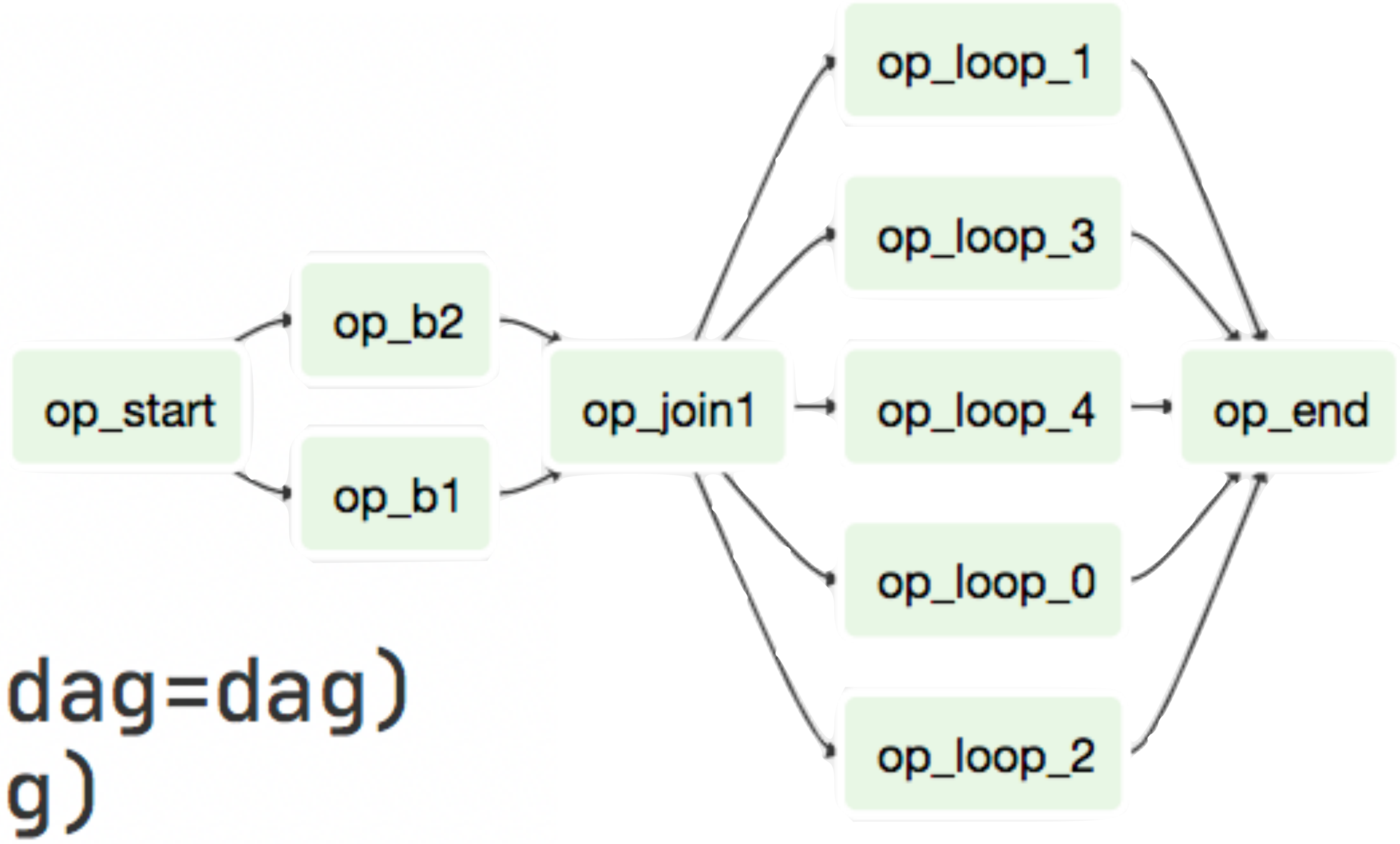
```
dag = DAG('o1_arglog_dag',
          description='Python Operator with Argument',
          start_date=datetime(2018, 1, 1),
          schedule_interval=None,
          catchup=False
          )

def arglog(*args, **kwargs):
    ds.dump_pp(kwargs, "kwargs.txt")
    ds.dump_pp(args, "args.txt")

arglog_op = PythonOperator(
    task_id='arglog_task',
    provide_context=True,
    python_callable=arglog,
    op_args=["a1", "a2"],
    op_kwargs={"k1": "v1", "k2": "v2"},
    dag=dag)

arglog_op
```

Backfill am Beispiel



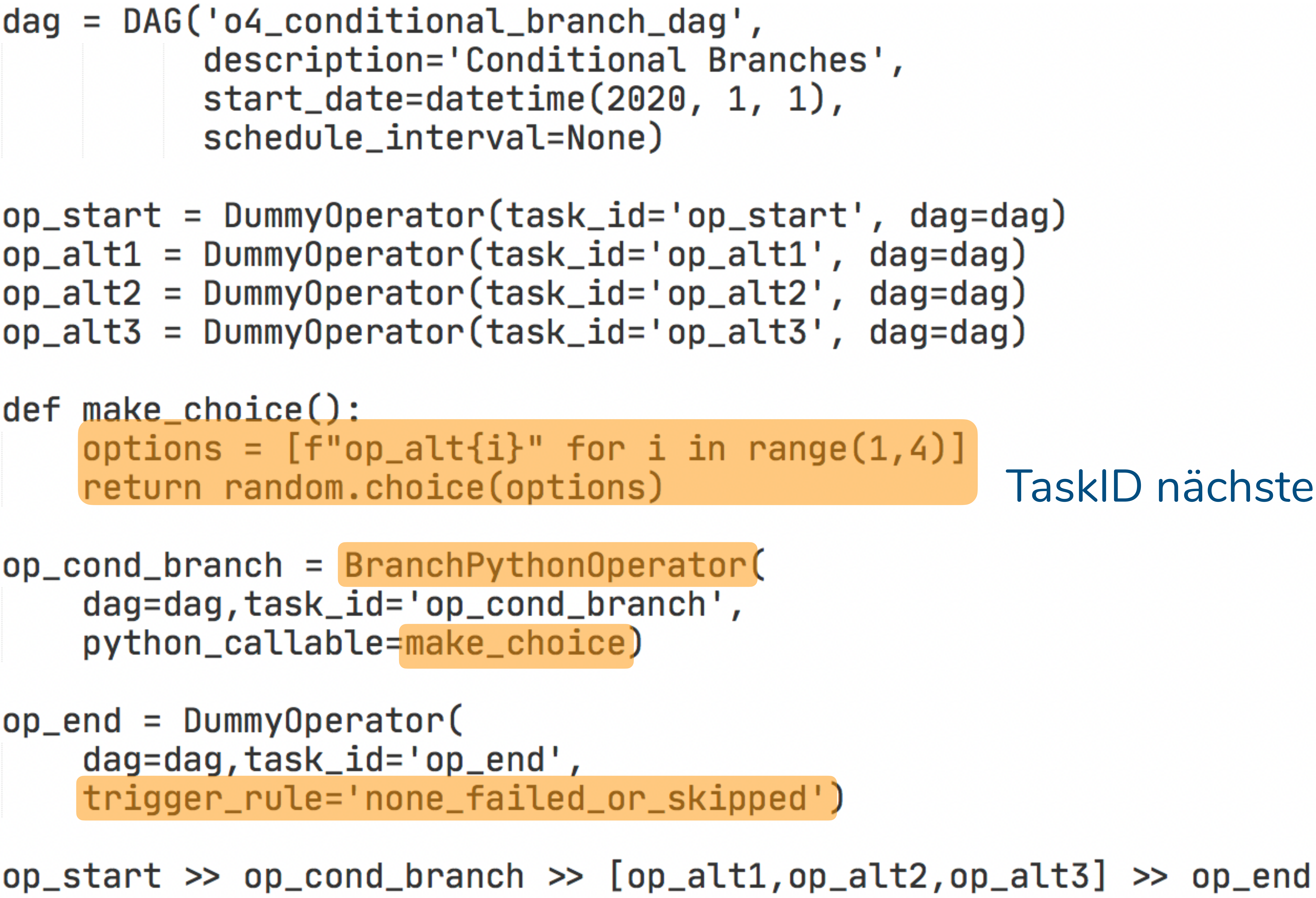
```
dag = DAG('02_backfill_dag',
          description='Python Operator with Backfill',
          start_date=datetime(2020, 1, 1),
          schedule_interval="@daily",)

def backfill(*args, **kwargs):
    p = "/Users/imhiro/Desktop/conferences_2020/4_enterpy/
    enterpy_2/code/data"
    date = kwargs['ds']
    url = f"http://localhost:5000/daily/{date}"

    res = requests.get(url)
    if res.ok:
        j = res.json()
        with open(os.path.join(p,f"res_{date}.json"),mode="w")
        as out:
            json.dump(j,out)
    else:
        raise Exception(f"Error {res.status_code}")

backfill_op = PythonOperator(
    task_id='backfill_task',
    provide_context=True,
    python_callable=backfill,
    dag=dag)

backfill_op
```

Fanout und Programmatische DAG-Erstellung


op_start
op_b2
op_b1
op_join1
op_loop_1
op_loop_3
op_loop_4
op_loop_0
op_loop_2
op_end

```
dag = DAG('loop_fan_dag',
          description='Loops and Fanout',
          start_date=datetime(2020, 1, 1),
          schedule_interval=None,
          )

op_start = DummyOperator(task_id='op_start', dag=dag)
op_b1 = DummyOperator(task_id='op_b1', dag=dag)
op_b2 = DummyOperator(task_id='op_b2', dag=dag)

op_join1 = DummyOperator(task_id='op_join1', dag=dag)

op_start >> [op_b1, op_b2] >> op_join1

op_end = DummyOperator(task_id='op_end', dag=dag)

for i in range(5):
    op = DummyOperator(task_id=f'op_loop_{i}', dag=dag)
    op_join1 >> op >> op_end
```

Bedingungen und Pfade



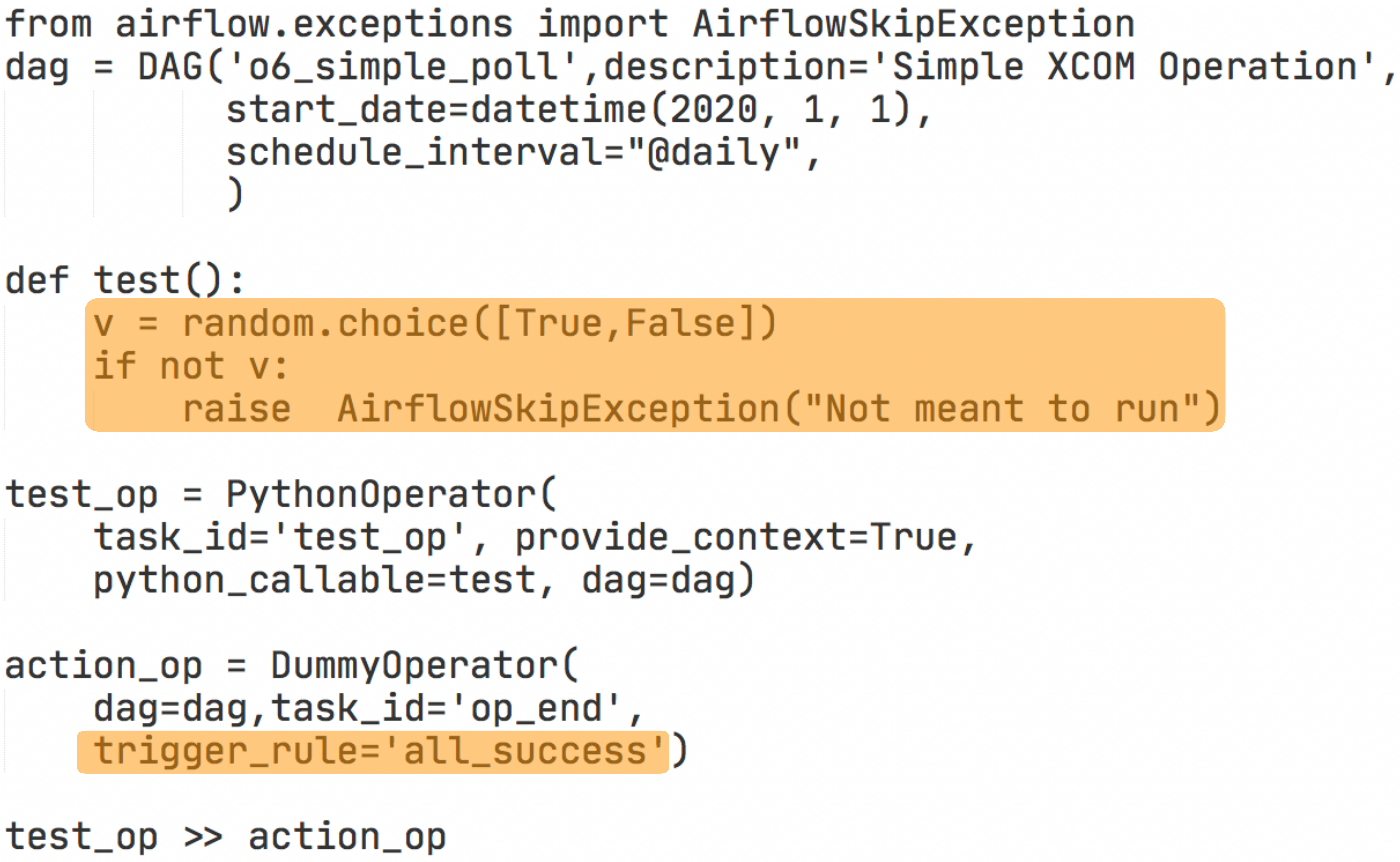
```
dag = DAG('o4_conditional_branch_dag',
          description='Conditional Branches',
          start_date=datetime(2020, 1, 1),
          schedule_interval=None)

op_start = DummyOperator(task_id='op_start', dag=dag)
op_alt1 = DummyOperator(task_id='op_alt1', dag=dag)
op_alt2 = DummyOperator(task_id='op_alt2', dag=dag)
op_alt3 = DummyOperator(task_id='op_alt3', dag=dag)

def make_choice():
    options = [f"op_alt{i}" for i in range(1,4)]
    return random.choice(options)

op_cond_branch = BranchPythonOperator(
    dag=dag,task_id='op_cond_branch',
    python_callable=make_choice)

op_end = DummyOperator(
    dag=dag,task_id='op_end',
    trigger_rule='none_failed_or_skipped')

op_start >> op_cond_branch >> [op_alt1,op_alt2,op_alt3] >> op_end
```

TaskID nächster Operator

Kommunikation mit XCOM

```
dag = DAG('o5_xcom_dag',
          description='Simple XCOM Operation',
          start_date=datetime(2020, 1, 1),
          schedule_interval="@daily")

def setter():
    return {"key": "value", "list": [1, 2, 3, 4]}

def getter(**context):
    ti = context['task_instance']
    value = ti.xcom_pull(key=None, task_ids='set_op')
    fp = "/Users/imhiro/Desktop/conferences_2020/4_enterpy/enterpy_2/code/data/xcom.txt"
    with open(fp, mode="w") as out:
        json.dump(value,out)

set_op = PythonOperator(
    task_id='test_op', provide_context=True,
    python_callable=setter, dag=dag)

get_op = PythonOperator(
    task_id='action_op', provide_context=True,
    python_callable=getter, dag=dag)

set_op >> get_op
```

Einfaches Polling



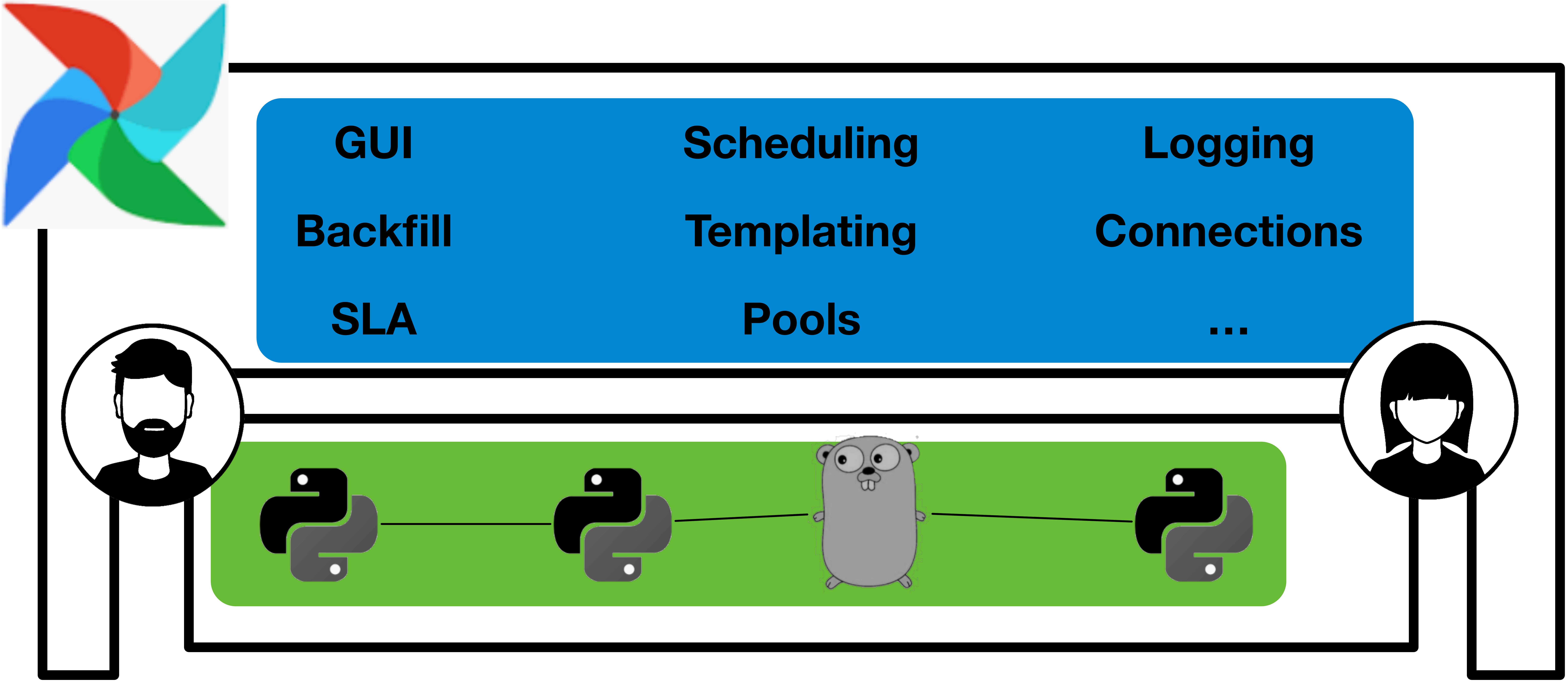
```
from airflow.exceptions import AirflowSkipException
dag = DAG('o6_simple_poll',description='Simple XCOM Operation',
          start_date=datetime(2020, 1, 1),
          schedule_interval="@daily",
          )

def test():
    v = random.choice([True,False])
    if not v:
        raise  AirflowSkipException("Not meant to run")

test_op = PythonOperator(
    task_id='test_op', provide_context=True,
    python_callable=test, dag=dag)

action_op = DummyOperator(
    dag=dag,task_id='op_end',
    trigger_rule='all_success')

test_op >> action_op
```

Fazit & Zusammenfassung

Fazit - Airflow Technik

- Airflow ermöglicht die Definition von komplexen Workflows
  - Gegenüber anderen Workflow-Lösungen oder selbst geschriebenem Scheduling bietet Airflow zahlreiche Standardfunktionen
  - Python dient als Sprache für die Definition von Workflows und kann zur Implementierung der einzelnen Workflowschritte verwendet werden. Dies macht die Verwendung mit Python Kenntnissen einfacher und fördert die Automatisierung
- Airflow skaliert in Bezug auf Deploymentmöglichkeiten und Funktionen von einem Rechner auf verteilte Unternehmensanwendungen

Fazit - Airflow und Python im Einsatz



GUI **Scheduling** **Logging**

Backfill **Templating** **Connections**

SLA **Pools** **...**

Vielen Dank