Going beyond with Jupyter notebooks

Welcome! This tutorial will present how to use Jupyter notebooks to publish python packages and more

Hello there

- Brazilian
- Computer scientist*
- RnD Python Developer
- Open source maintainer
- Python community contributor (Grupy RN)

grupy.rn

Hello there

This tutorial will:

- Introduce the main concepts to publish a package and why to do it
- Discuss the Python packaging system and distribution
- Explain the literate programming concept
- Show Jupyter notebook and Jupyter lab UI
- Introduce Nbdev lib and how we can use literate programming to publish Python packages
- Explain good practices in notebooks and Nbdev
- Develop a French Deck lib and mini game

Hello there

- Display quality assurance tools
- Discuss notebook testing
- Show how and where to publish a Python package
- Introduce Github Actions usage add a CI/CD to our workflow
- Talk about Python packaging past and future
- Introduce how to build UI for notebooks
- Show Quarto customization
- Demonstrate how to use ChatGPT in notebooks

Python packaging

Let’s introduce/review some core concepts to understand python packaging and its structure

Modules

> Modules are files that contain Python definitions and statements, Python Docs

Any `.py` file can be considered a module

```
%%writefile module.py

def hello(name):
    print(f'hello, {name}')
```

```
import module

module.hello("audience!")
```

Modules

Modules can also be executed as python scripts

```
%%writefile module.py

def hello(name):
    print(f'hello, {name}')

if __name__ == "__main__":
    import sys
    hello(sys.argv[1])
```

```
! python module.py 'Maria'
```

> **Note**
>
> The conditional `if __name__ == "__main__"` makes sure that the function `hello` will be executed only when the module is running as a “main” file. This strategy allows python files to be imported and also used as scripts

Python modules search

Python interpreter searchs for a module:

- Looking first at its built-in modules (listed at `sys.builtin_module_names`)
- Looks for python files in the directory list (`sys.path`)

Packages

> Is a way to struct Python namespaces using dotted module names, Python Docs

Packages can be understood as a collection of modules. Check the following structure.

```
somefolder/
    {username}_package/
        __init__.py
        module1.py
        subpackage/
            module2.py
```

Note

- Create this folder structure on your machine
- Open your terminal
- Import module1 `from {username}_package import module1`
- Import module2 `from {username}_package.subpackage import module2`
- Access a different folder (`cd ..`) and import {username}_package
- `import sys; sys.path.append('somefolder'); import {username_package}`

Python package search

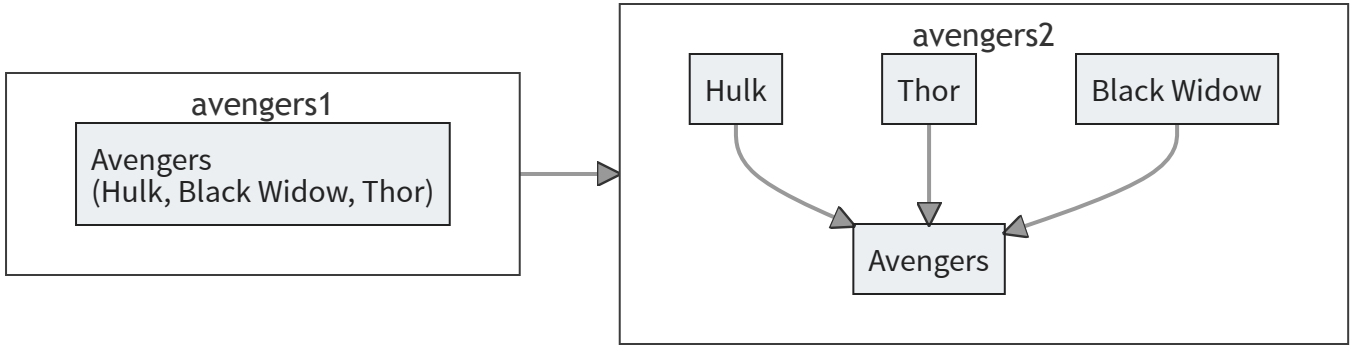
When importing packages, Python looks for the directories names listed at `sys.path`

Why packaging?

Let’s discuss the pros and cons of packaging

Why packaging?

Let's think about the "Avengers" movie as a big package with multiple heroes (Hulk, Black Widow, Thor, etc).



Note

Example from Arquitetura Modular com pacotes Python talk presented at Python Brazil, 2022

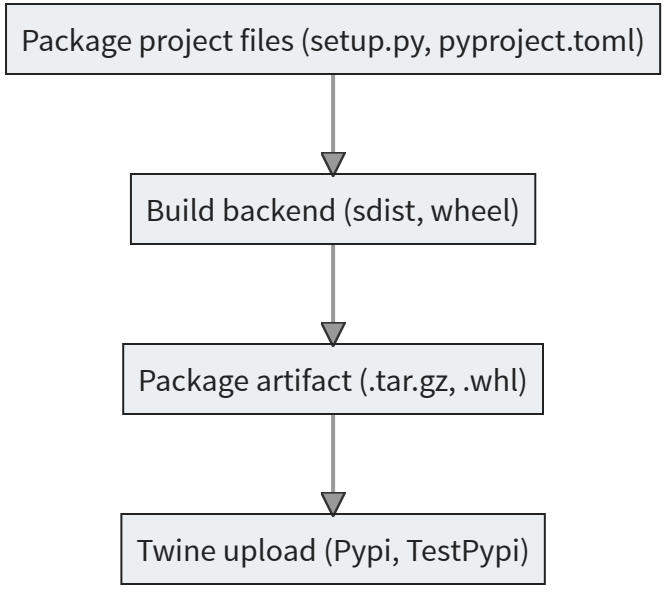
Pros and cons

- Increases the code reuse
- Reduces coupling
- Easier for other developers to use
- Split responsability across teams
- It can make maintenance easier

- Dependency management can be difficult
- Security of third-party packages
- Packages sometimes have more features than what's required

How to publish?

How can we publish the package in PyPI and download it?



Pypi

Python Package Index

Tooling

- pip is the most popular package manager in python
- PyPI is the index where pip downloads content from
- Test PyPI a separated index for testing

> **Note**
>
> Create your account at https://test.pypi.org, let's use it to publish our {username} package

Publishing {username} package

- Go to the root (`somefolder`) of the “{username} package”
- Create a setup.py file with the following content:

```
from setuptools import setup

setup(
    name='{username}_package',
    version='0.0.1',
    packages=['{username}_package']
)
```

- Create a virtualenv `python -m venv minimal_package && souce minimal_package/bin/activate`
- Run `python setup.py sdist` and `python setup.py bdist_wheel`
- Check the `dist` folder

Publishing {username} package

- Check if twine is available by running `twine` in your terminal, otherwise run `pip install twine`
- (Make sure your Test PyPI email is verified)
- Run `twine upload --repository-url https://test.pypi.org/legacy/ dist/*`
- Check your Test Pypi account

Other packaging repositories

There are others repositories than PyPI and Test PyPI

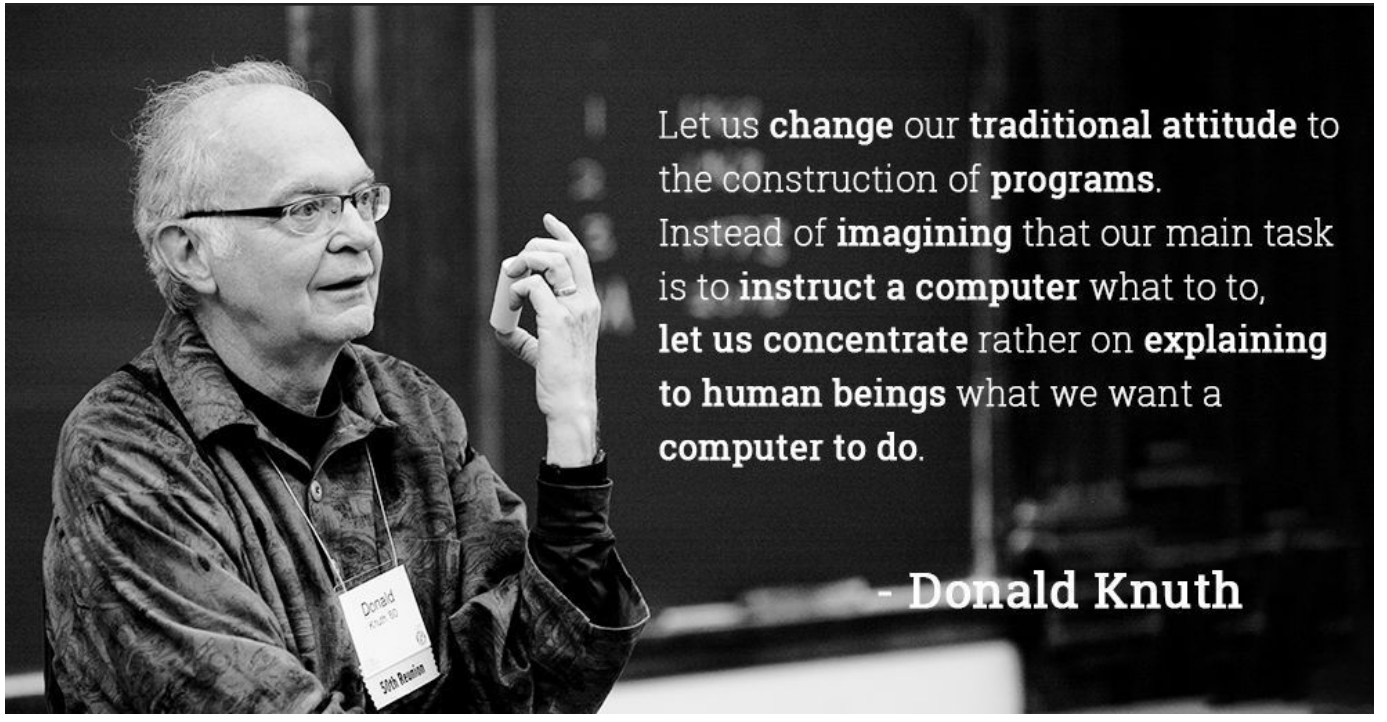
- Pypiserver local or self-hosted
- JFrog self-host or cloud, complete cloud solution
- Code artifacts AWS cloud repository
- Artifact registry Google cloud repository
- Gitlab package registry Gitlab cloud repository, excelent to use in private projects

Literate programming

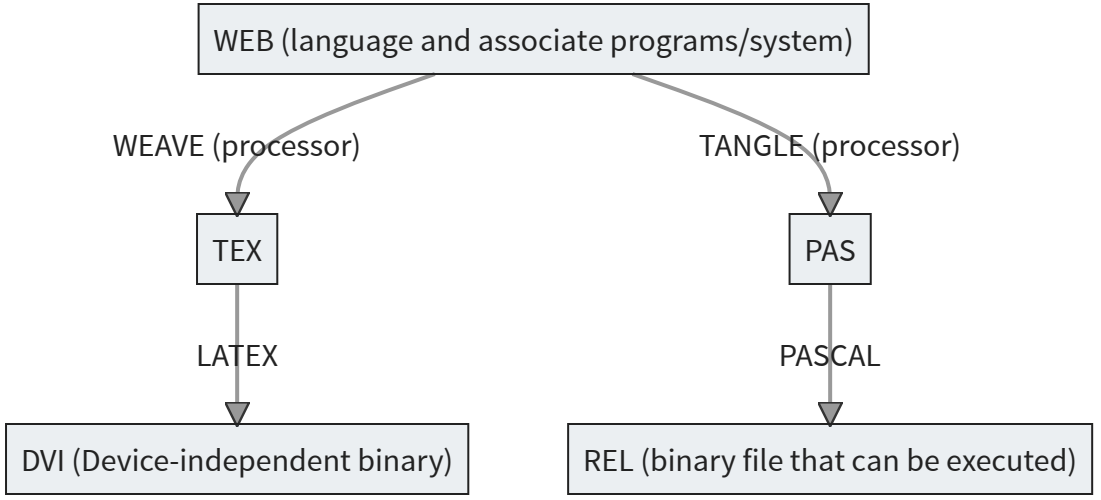
Programming paradigm in which your code tells a story

Literate programming

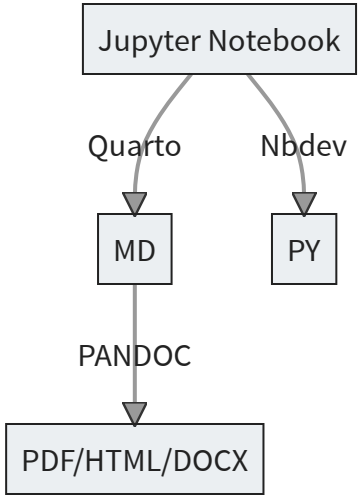
- Programming paradigm
- Haskell `.lhs` vs `.hs`
- Usage increased in the last years
- Jupyter notebooks, R Studio



Literate programming



Literate programming



Jupyter notebook introduction

- Using the virtualenv created at {username}_package, install jupyter notebooks: `pip install notebook`
- Run `jupyter-notebook`
- Let's get familiar with the UI (shortcuts, cell nav, cell types, .ipynb format)

UI

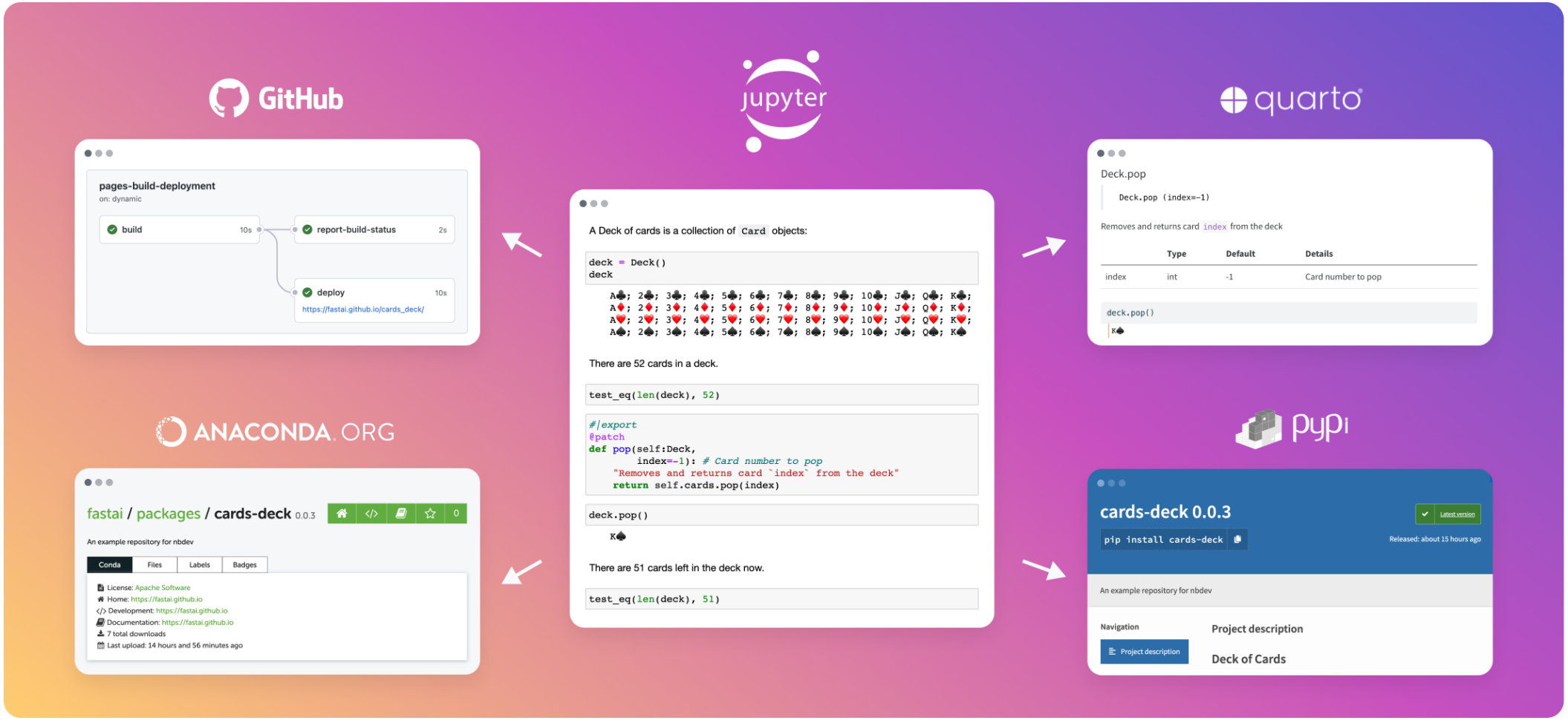
- Jupyter Lab it's one of the most popular UI
- Run `pip install jupyterlab` and execute `jupyter-lab` to see its interface

There are additional UIs, like the VSCode plugin that renders notebooks, but I recommend you to use classic notebooks in this tutorial

Nbdev

Using Jupyter notebooks to publish Python packaging

Nbdev


GitHub
pages-build-deployment
on: dynamic
build 10s
report-build-status 2s
deploy 10s
https://fastai.github.io/cards_deck/
jupyter
A Deck of cards is a collection of Card objects:
deck = Deck()
deck
There are 52 cards in a deck.
test_eq(len(deck), 52)
#|export
@patch
def pop(self:Deck,
index=-1): # Card number to pop
"Removes and returns card `index` from the deck"
return self.cards.pop(index)
deck.pop()
There are 51 cards left in the deck now.
test_eq(len(deck), 51)
quarto
Deck.pop
Deck.pop (index=-1)
Removes and returns card index from the deck
Type
Default
Details
index
int
-1
Card number to pop
deck.pop()
ANACONDA.ORG
fastai / packages / cards-deck 0.0.3
An example repository for nbdev
Conda
Files
Labels
Badges
License: Apache Software
Home: https://fastai.github.io
Development: https://fastai.github.io
Documentation: https://fastai.github.io
7 total downloads
Last upload: 14 hours and 56 minutes ago
pypi
cards-deck 0.0.3
pip install cards-deck
Latest version
Released: about 15 hours ago
An example repository for nbdev
Navigation
Project description
Project description
Deck of Cards

Install

- Let's move on from our {username}_package project and fork the following repository: https://github.com/itepifanio/going-beyond-with-jupyter-notebooks
- Create a new virtualenv `python -m venv tutorial`
- Activate your new virtualenv `source ./tutorial/bin/activate`
- Install the package running `pip install -e .[dev]`
- nbdev_install_hooks
- nbdev_install_quarto
- Run `python ./scripts/rename_package.py {username}_deck`

> **Note**
>
> Let's get familiar with the nbdev structure (settings.ini, nbs/, setup.py)
>
> Run `nbdev_help` in your terminal and check all available commands

Nbdev magic comments

- `#| hide`
- `#| export`
- `#| exporti`
- `#| exec_doc`
- `#| code-fold`
- `#| default_exp core`
- `#| eval: false`

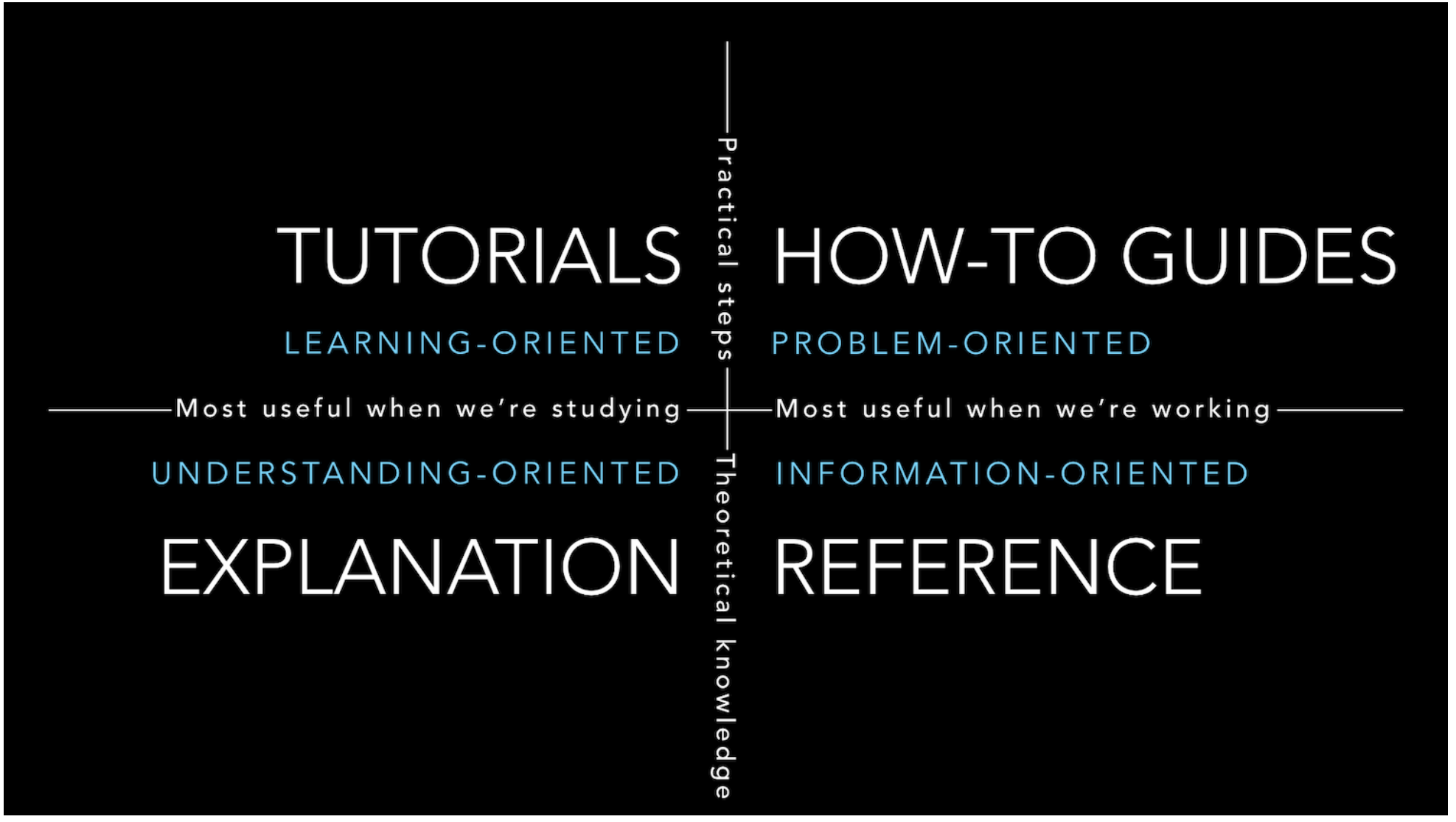
Nbdev awesome projects

- FastAI
- Ipyannotator
- TSAI
- FastKafta
- Number Blog
- UPIT
- AskAI
- Streamlit Jupyter
- Banet

Good practices

Jupyter notebook and nbdev best practices

Notebook type



Know the type of notebook you're writing (Diataxis System)

Good title and subtitle

There are two ways of doing that, one with markdown:

```
# My H1 title

> My description
```

Other using Quarto:

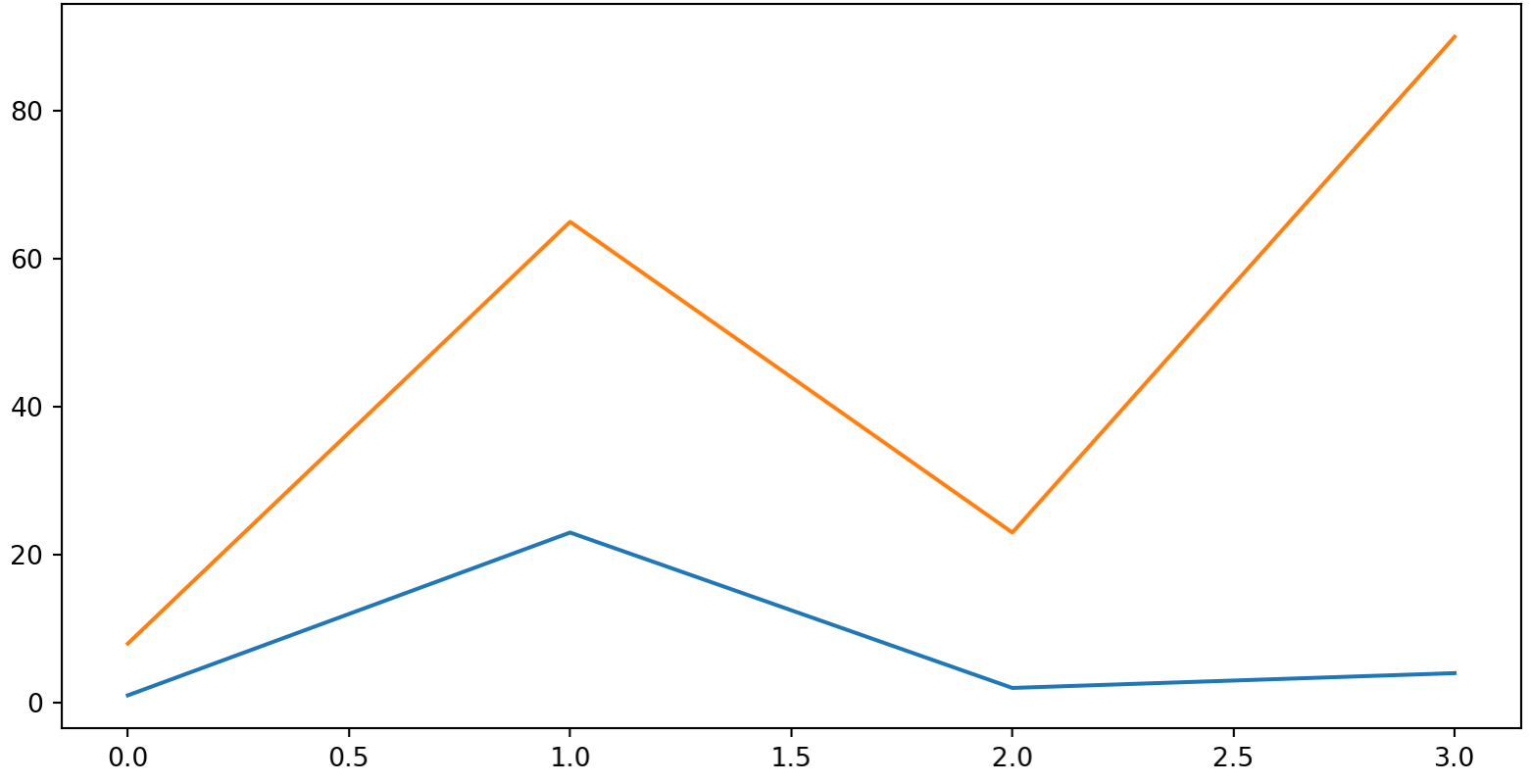
```
---
title: "My title"
description: "My description"
---
```

Change the text according to the notebook type

- Reference: Start with a small description of the component, use symbolic links for easy navigation
- Tutorials and guides: Describe what the reader will learn and how. Be objective
- Explanation: A brief review about the topic is enough to guide the reader

Use visualizations

Jupyter notebooks are very interactive. Use images, videos, audio



Keep your cells small

- There are no rules for cell size, but take it easy
- Use @patch `from fastcore.basics import patch`
- Increase the notebook's testability by writing small function/classes per cell

Jupyter notebooks

- Keep your imports at the top of the notebook
- Don’t import and use code at the same cell
- Avoid ambiguous execution orders
- Use code cells for experiments

Doc parameters

Nbdev has two types of parameter documentation, the classic (numpy-style), as it follows:

```
def add_np(a, b=0):
    """The sum of two numbers.

    Parameters
    ----------
    a : int
        the 1st number to add
    b : int
        the 2nd number to add (default: 0)
    """
    return a + b
```

Doc parameters

add_np

```
add_np (a, b=0)
```

The sum of two numbers.

	Type	Default	Details
a	int		the 1st number to add
b	int	0	the 2nd number to add (default: 0)

Doc parameters

And the nbdev approach called “docments”

```python
def add(
    a: int,  # the 1st number to add
    b=0,  # the 2nd number to add
):
    "The sum of two numbers."
    return a + b
```

---

add

> `add (a:int, b=0)`

The sum of two numbers.

	Type	Default	Details
a	int		the 1st number to add
b	int	0	the 2nd number to add

Write tests

- Keyword assert can be used for testing
- Run `nbdev_test`

```
1 assert add(1, 1) == 2
```

Exercise

```
1  import collections
2
3  Card = collections.namedtuple("Card", ["rank", "suit"])
```

Note

- Create a new notebook called `card_utils.ipynb`
- Add a title and description to the notebook
- Copy the `Card` class definition
- Create a function `card_name` that returns `1 of Hearts`
- Test that the `card_name` function returns the expected outputs
- Add your notebook to `sidebar.yml`
- Run `nbdev_preview` and see the changes
- Run `nbdev_test` and check if the tests are passing

Package

We're already familiar with the `Card` definition. Let's investigate the package code.

Exercise

> **Note**
>
> - Add a code cell at the top of the `card_utils.ipynb` nb with `#| default_exp card_utils`
> - Next, add a code cell with `from nbdev import nbdev_export` (remember adding a `#| export` comment to the `def card_utils(...)` cell)
> - At the end of `card_utils.ipynb` add `nbdev_export()`
> - Execute nb and check the `{username}_deck` folder

Dependencies

Avoid dependency hell for others

Dependency Hell

- When a software grows, more libs tend to be added
- Packages are awesome: they can add features, avoid/fix errors and provide more security to a software
- But keeping packages updated can be tricky
- Ex. Dependency 1 expects Python 3.7 usage and Dependency 2 expects Python 3.9

Specify your dependencies

- ~= to specify releases
- == to fix a version
- != to exclude a version
- <=, >= to inclusively specify a range of versions
- <, > to specify a range of versions

Releasing

- Semantic versioning is a set of rules that helps avoiding dependency hell
- It MUST declare a public API and it SHOULD be comprehensive
- The version number MUST use the X.Y.Z format where X, Y, Z are non-negative integers and each element SHOULD increase numerically
- MAJOR.MINOR.PATCH

Keep a changelog

- Semantic versioning is very didatic but it’s not human readable
- Changelog keeps a chronological order list of notable changes of each version of a project
- It improves the transparency
- Helps keeping developers accountable
- Helps stakeholders to understand the project’s direction
- Keeps a list of the bugs

Quality Assurance

Improve your code by using automatic tools

NbQA

There are several tools that analise code and:

- Avoid errors and code smells
- Define a code style
- Improve readability
- Measure code quality

NbQA

Popular tools that analise code:

- Autopep8
- Black
- Flake8
- MyPy
- Isort

All of them (and some others) can be executed using NbQA

```
1 nbqa <tool-name> <tool-params>
```

MyPy

> Static type checker. It validates your code typing without executing it

Previous to Python 3.5 (we didn't have typing), it was common to use docstring to define types:

```
def natural_sum(x, y):
    """
    Args:
        x (int): first parameter of the sum
        y (int): second parameter of the sum
    Returns:
        Optional[int]: Sum of x and y if both bigger than zero
    """
    if x < 0 or y < 0:
        return None

    return x + y
```

MyPy

The need of tools to validate data typing made Python add the typing system to its core.

Run the following snippet on your terminal:

```
! nbqa mypy *.ipynb --ignore-missing-imports
```

Note

The type system was developed to help other tools like MyPy to check types statically but libs like pydantic started to validate typing in runtime as well for some use cases

Tip

Check MyPy usage creating a adding function:

```
def add(a: int, b: int) -> int:
  return a + b
```

and testing the function using:

```
add('1', '2')
```

Flake8

> Is a style guide enforcement

It saves time when a team starts discussing empty spaces and small code style details on pull requests

Run the following snippet in your terminal:

```
! nbqa flake8 *.ipynb
```

Tip

- Test flake8 usage by adding a import that it's not used like `import matplotlib`
- Check the .flake8 file (it is used by autopep8* as well)

Autopep8

> Code format enforcer according to PEP 8

It’s very useful to clean empty space and format notebooks with a well known code style

Run the following snippet in your terminal:

```
! nbqa autopep8 nbs/*.ipynb --in-place
```

Black

> Another code format enforcer but more flexible

Works similarly to autopep8 but it's not restricted to PEP 8

Run the following snippet in your terminal:

```
! nbqa black nbs/*.ipynb
```

Advanced testing

Test isolation with i(pytest). Let’s explore the advanced testing nb

Publishing

Nbdev allows publish package by using its CLI

Conda and PyPI

- It can publish on both: `nbdev_pypi`, `nbdev_conda`, `nbdev_release_both`
- Less documented but it also allows Test PyPI usage (`nbdev_pypi --repository testpypi`)

> **Note**
>
> As we did before with the `{username}_package` let's publish this package using `nbdev_pypi --repository testpypi`

Secret keys

Python package system uses the .pypirc to define the repository packages:

```
[distutils]
index-servers = testpypi

[testpypi]
repository=https://test.pypi.org/legacy/
username=<your-username>
password=<your-password>
```

Secret keys

Username and password usage are not encouraged, consider generating a token for your package:

```
[distutils]
index-servers = testpypi

[testpypi]
repository=https://test.pypi.org/legacy/
username = __token__
password = <PyPI token>
```

Custom repositories

To add custom repositories (private or not) you can change `~/.pypirc` and add the repository name to `[distutils]`

```
[distutils]
index-servers =
    pypi
    testpypi
    private-repository

[pypi]
username = __token__
password = <PyPI token>

[testpypi]
username = __token__
password = <TestPyPI token>

[private-repository]
repository = <private-repository URL>
username = <private-repository username>
password = <private-repository password>
```

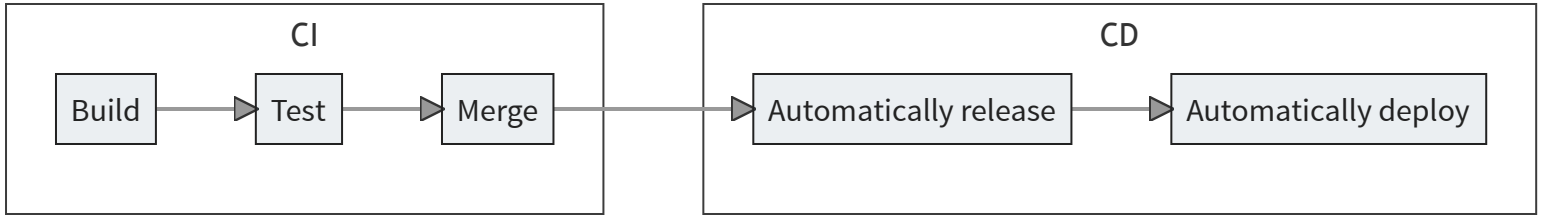
CI/CD

Let’s run QA and publish our package using Github Actions

Continuous Integration/Continuous Delivery

- Well known concept in DevOps that recently became more acessible to developers
- Method to frequently deliver apps to customers
- It automates some stages of the app development
- Enforces security of the deliver

Continuous Integration/Continuous Delivery



Tools

- Github Actions
- Gitlab CI/CD
- Jenkins
- Circle CI

And many others

Github Actions

- Easy to use
- Free until 2000 minutes/month
- Most tools have some integration with it

Github actions core concepts

- *Events*: It's a specific activity in your Github repository that triggers an action. For example, the opening of a PR, a commit being sent, etc
- *Jobs*: It's a sequence of steps that will be executed by a shell script or action. Jobs can be executed in parallel or sequentially
- *Action*: Custom application that uses the Github Action platform, it usually automates a repetitive task like the configuration of an environment or managing a complex dependency, even service authentication (like cloud)
- *Runner*: Server that executes workflows. Every runner executes a job per time

Github secrets & Test Pypi token

- Let’s create a token for our {username}_deck repository

> **Note**
>
> At Test PyPI webpage access `Account Settings >> Api Tokens >> Add Api Token`

- In your Github repository go to: `Settings >> Actions (at security tab) >> New repository secret` and add the token with the name `TEST_PYPI_API_TOKEN`

Workflows

(Interactive explanation of files from `.github/worflows/*.yaml`)

Publishing using CI

Note

- Upgrade the version at `settings.ini`
- Update your changelog
- Commit and push your changes to Github
- Check if tests and lint are passing
- Create a new release at Github following the semantic versioning to use our CD of the package

Python Packaging history

Let’s look at the past to discuss the future of Python packaging

History

- Python 1 (1998-2000) didn’t have a package manager
- Distutils was added to Python 1.6 using setup.py
- In 2003 setuptools was introduced as an improvement of distutils
- In 2004 easy_install was developed to be used alongside setuptools
- In 2008 the PyPA (Python Packaging Authority) was founded

History

- In 2011 `pip` became the default package manager
- In 2013 the `wheel` package was introduced
- In 2017 the `flit` package was developed, introducing `pyproject.toml`
- In 2020 the PEP 621 made `pyproject.toml` the standard way to develop packages

Quarto and Styling

Quarto is an open source tool that allows creating content dinamically using Python, R, Julia and Observable. It was introduced to Nbdev recently

Executing

Try to execute `quarto preview nbs/00_deck.ipynb` in your terminal

> **Note**
>
> Quarto is already installed on our setup thanks to `nbdev_install_quarto`

Doc preview

Changing ports when rendering nbdev docs can be annoying, change `_quarto.yml` to fix a port and avoid opening new tabs.

```
project:
  preview:
    port: 3000
    browser: false
```

Doc navigation

Quarto allows easy customization of the navbar

```yaml
website:
  navbar:
    background: primary
      search: true
      collapse-below: lg
      left:
        - text: "My page"
          href: gettings_started.ipynb
      right:
        - icon: github
          href: "https://github.com/user/project"
```

Google analytics

Activate analytics tracking but remember asking for cookie consent if your country legislation requires it

```
website:
  google-analytics: "UA-XXXXXXXX"
  cookie-consent: true
```

Dark mode

You can define the quarto theme to (de)activate dark mode

```yaml
format:
  html:
    theme:
      light: flatly
      dark: darkly
```

Page navigation

Your project may require continuous page navigation

```
website:
  page-navigation: true
```

Reader mode

Enable reader mode

```
website:
  reader-mode: true
```

Playground

We've already published our package and know more about nbdev powers, let's have some fun with nbs

Ipywidgets

Let’s run the UI for jupyter notebooks nb

> **Tip**
>
> Run voila over the `03_ui_for_jupyter_notebook.ipynb` using `voila 03_ui_for_jupyter_notebook.ipynb`

ChatGPT

Let’s use ChatGPT in our nbs by running the ChatGPT nb

Conclusion

- Jupyter notebooks are awesome for quick prototyping
- Code, doc and tests can be used to tell a story and improve docs
- Jupyter notebooks & nbdev allow literate programming
- Publishing a package with nbdev is easy, faster and efficient
- Visual libraries benefit a lot from nbdev