

Type hints

CREATING ROBUST PYTHON WORKFLOWS



Martin Skarzynski

Co-Chair, Foundation for Advanced
Education in the Sciences (FAES)

Dynamic typing

```
def double(n):  
    return n * 2
```

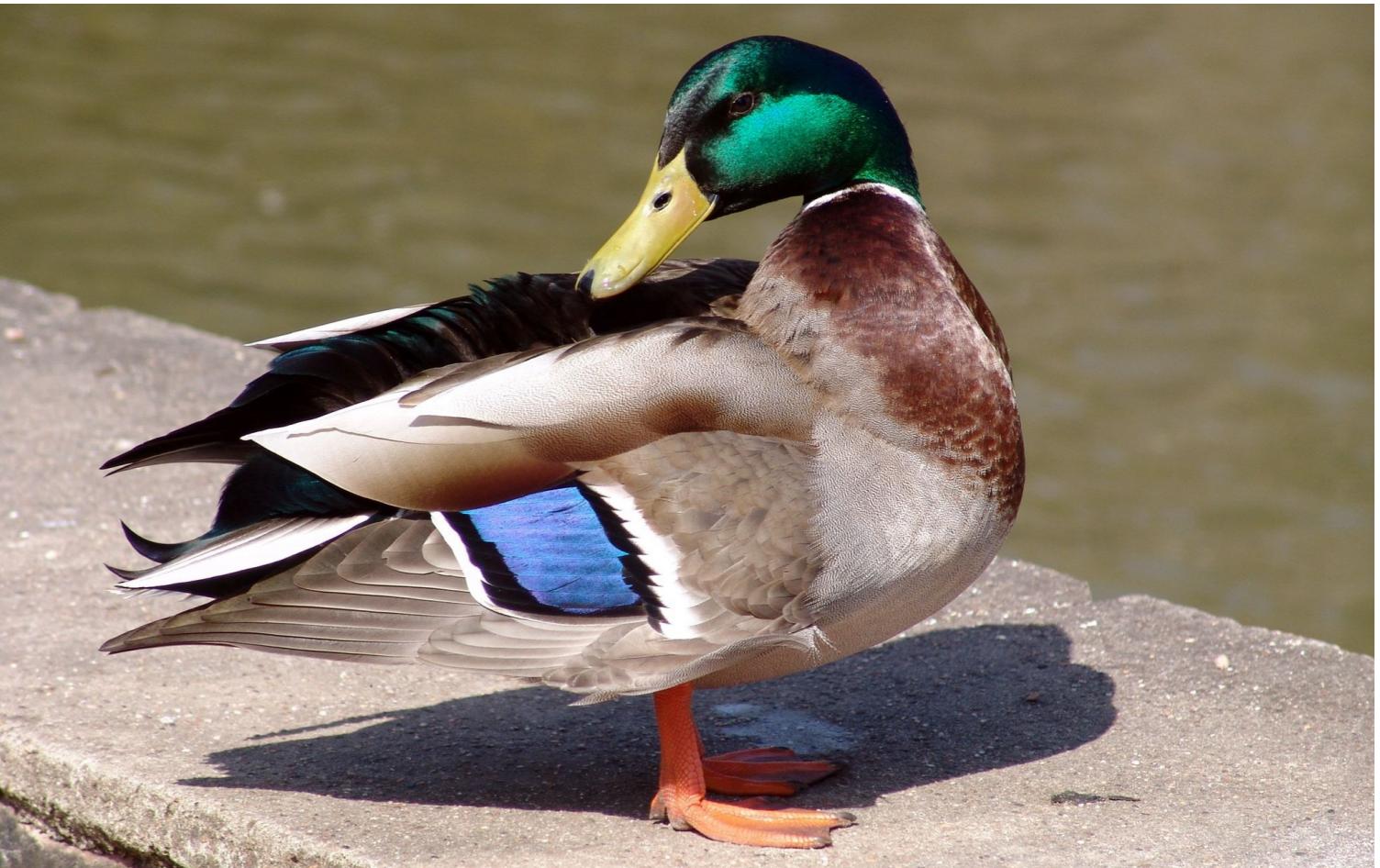
```
double(2)
```

```
double('2')
```

```
4
```

```
'22'
```

- Python
 - Infers types when running code
 - Dynamic (duck) typing



Type hints for arguments

```
def double(n: int):  
    return n * 2  
  
double(2)
```

4

```
def double(n: str):  
    return n * 2  
  
double('2')
```

'22'

Type hints for return values

```
def double(n: int) -> int:  
    return n * 2  
  
double(2)
```

4

```
def double(n: str) -> str:  
    return n * 2  
  
double('2')
```

'22'

Get type hint information

```
from double import double
```

```
# The help() function  
help(double)
```

```
Help on function double in module double:
```

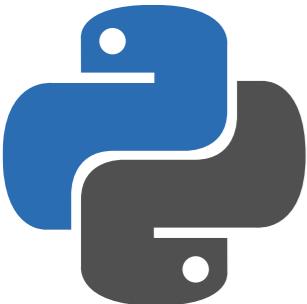
```
double(n:int) -> int
```

Type checker setup

Type checking tool setup:

- mypy type checker
- pytest testing framework
- pytest-mypy pytest plugin

```
$ pip install pytest mypy pytest-mypy
```

 : my[py]



pytest

Type checker setup

- `pytest.ini` file with the following:

```
[pytest]
addopts = --doctest-modules --mypy --mypy-ignore-missing-imports
```

Mypy to the rescue!

```
$ pytest double.py
```

```
===== test session starts =====
...
=====
FAILURES
=====
----- mypy double.py -----
double.py:4: error: Arg. 1 to "double" has incompatible type "str"; expected "int"

=====
1 failed in 0.36 seconds =====
```

List

```
from typing import List

def cook_foods(raw_foods: List[str]) -> List[str]:
    return [food.replace('raw', 'cooked') for food in raw_foods]

cook_foods(['raw asparagus', 'raw beans', 'raw corn'])
cook_foods('raw corn')
```

```
['cooked asparagus', 'cooked beans', 'cooked corn']
['r', 'a', 'w', ' ', 'c', 'o', 'r', 'n']
```

Pytest cook

```
$ pytest cook.py
```

```
===== test session starts =====
...
=====
FAILURES
=====
----- mypy cook.py -----
cook.py:7: error: Arg. 1 to "cook_foods" has ... type "str"; expect. "List[str]"
=====
1 failed in 0.25 seconds =====
```

Optional

```
from typing import Optional

def str_or_none(optional_string: Optional[str] = None) -> Optional[str]:
    return optional_string
```



Let's practice type annotating our code!

CREATING ROBUST PYTHON WORKFLOWS

Docstrings

CREATING ROBUST PYTHON WORKFLOWS



Martin Skarzynski

Co-Chair, Foundation for Advanced
Education in the Sciences (FAES)

Docstrings

- Triple quoted strings
- Include documentation in objects

```
def double(n: float) -> float:  
    """Multiply a number by 2."""  
    return n * 2
```

'''

Access docstrings

```
help(double)
```

```
Help on function double in module __main__:
```

```
double(n: float) -> float
```

```
Multiply a number by 2.
```

Google docstring style

```
"""Google style.
```

The Google style tends to result in wider docstrings with fewer lines of code.

Section 1:

- Item 1: Item descriptions don't need line breaks.

```
"""
```

Numpy docstring style

```
"""Numpy style.
```

The Numpy style tends to results in narrower docstrings with more lines of code.

Section 1

Item 1

Item descriptions are indented on a new line.

```
"""
```

Docstring types

```
"""MODULE DOCSTRING"""

def double(n: float) -> float:
    """Multiply a number by 2."""
    return n * 2

class DoubleN:
    """CLASS DOCSTRING"""
    def __init__(self, n: float):
        """METHOD DOCSTRING"""
        self.n_doubled = n * 2
```

- Location determines the type:
 - In definitions of
 - Functions
 - Classes
 - Methods
 - At the top of .py files
 - Modules
 - Scripts
 - __init__.py

Package docstrings

```
import pandas  
help(pandas)
```

Help on package pandas:

NAME
pandas

DESCRIPTION
pandas - a powerful data analysis
and manipulation library for Python

help() output highlights:

- NAME
- DESCRIPTION (package docstring)
- FILE (path to `__init__.py`)

Module docstrings

```
import double  
help(double)
```

Help on module double:

NAME
double - MODULE DOCSTRING

CLASSES
builtins.object
DoubleN

```
class DoubleN(builtins.object)  
| DoubleN(n: float)  
|  
| CLASS DOCSTRING  
|  
| Methods defined here:  
|  
| __init__(self, n: float)  
| METHOD DOCSTRING
```

Class docstrings

```
class DoubleN:  
    """The summary of what the class does.  
  
    Arguments:  
        n: A float that will be doubled.  
  
    Attributes:  
        n_doubled: A float that is the result of doubling n.  
    """  
  
    def __init__(self, n: float) -> None:  
        self.n_doubled = n * 2
```

Docstring examples

```
def double(n: float) -> float:  
    """Multiply a number by 2.  
  
    Arguments:  
        n: The number to be doubled.  
  
    Returns:  
        The value of n times 2.  
  
    Examples:  
        >>> double(2)  
        4.0  
    """  
  
    return n * 2
```

Mistake in the docstring example:

2 * 2

4

2. * 2

4.0

Test docstring examples

```
===== FAILURES =====
----- [doctest] double.double -----
005     Returns:
006         The value of n times 2.
007     Examples:
008         >>> double(2)

Expected:
4.0

Got:
4

MODULE/square.py:8: DocTestFailure
==== 1 failed, 1 passed in 0.26 sec. ===
```

```
$ pytest double.py
```

- Docstring examples combine
 - Documentation
 - Tests (via `doctest`)

Module docstring examples

```
"""Module docstring
```

Examples:

```
>>> dn = DoubleN(2)
```

```
>>> dn.n_doubled == double(2)
```

```
True
```

```
"""
```

```
def double(n: float) -> float:
```

```
    return n * 2
```

```
class DoubleN:
```

```
    def __init__(self, n: float):
```

```
        self.n_doubled = n * 2
```

```
$ pytest double.py
```

```
===== test session starts =====
```

```
...
```

```
double.py ..
```

```
[100%]
```

```
===== 2 passed in 0.36 seconds =====
```

Let's practice writing docstrings!

CREATING ROBUST PYTHON WORKFLOWS

Reports

CREATING ROBUST PYTHON WORKFLOWS

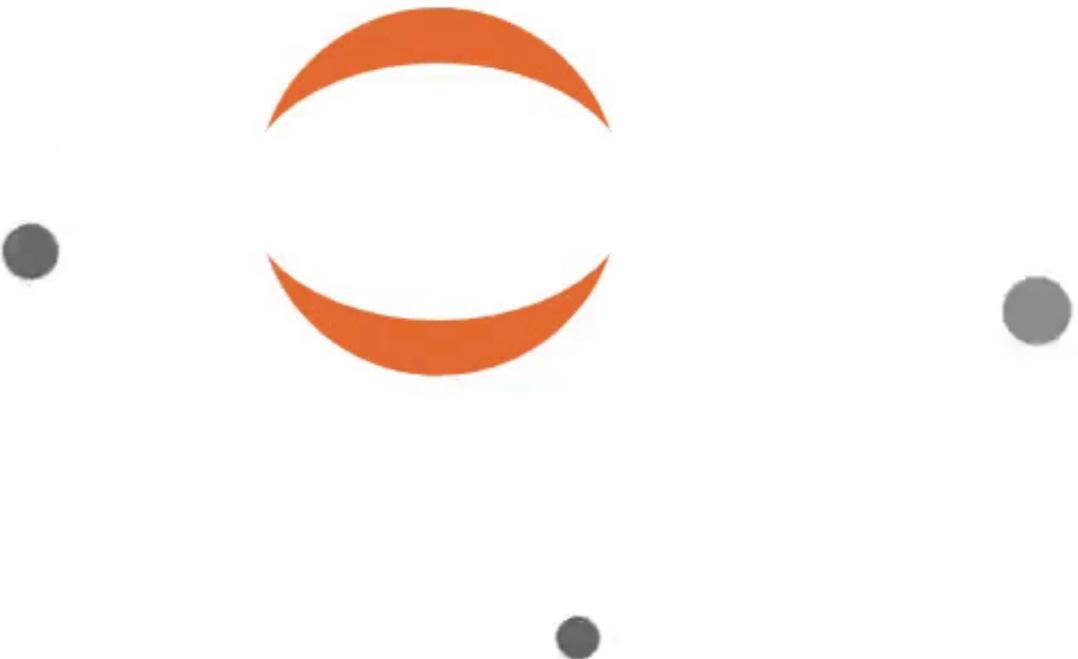


Martin Skarzynski

Co-Chair, Foundation for Advanced
Education in the Sciences (FAES)

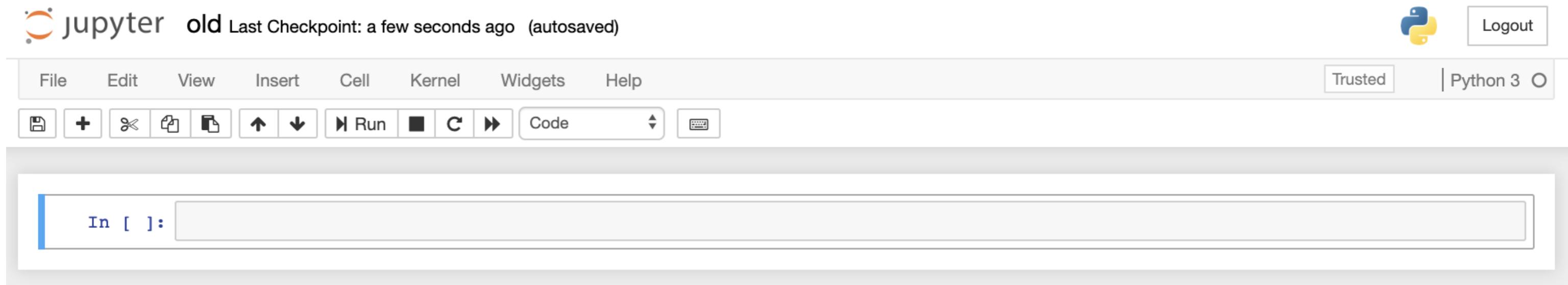
Jupyter notebooks

- Consist of cells
 - Text (Markdown format)
 - Code (Python, R, etc.)
- Have an `.ipynb` extension
- Built on IPython
- Have a structure based on JSON
 - JavaScript Object Notation
 - Similar to a Python dictionary



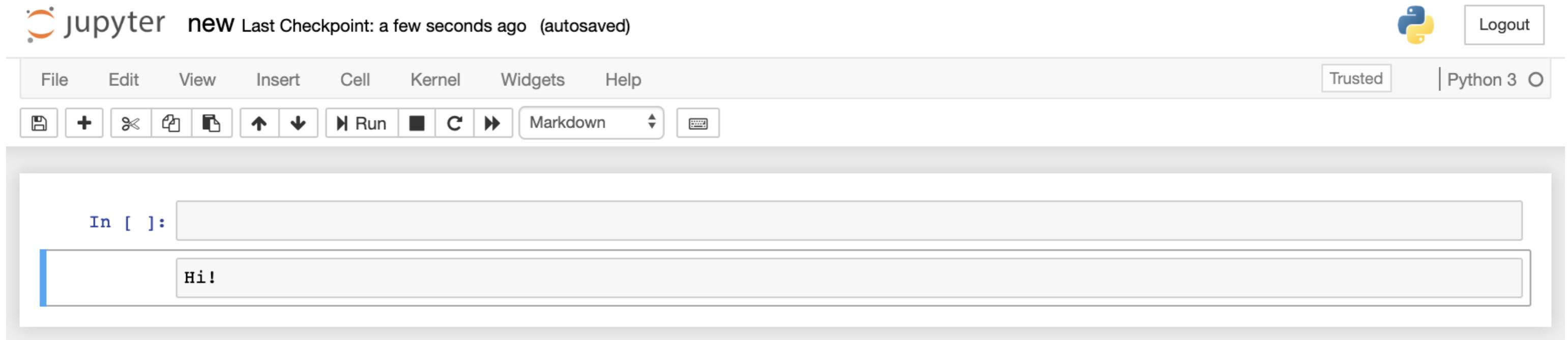
¹ Pérez, F., & Granger, B. E. (2007). IPython: a system for interactive scientific computing. CiSE, 9(3).

Track notebooks changes



- old.ipynb
 - Empty code cell

Track notebooks changes



- new.ipynb
 - Empty code cell
 - Markdown cell that says Hi!

Diff

- View changes made to notebooks
 - With the `diff` shell command

```
$ diff -c old.ipynb new.ipynb
```

```
"source": []  
+ },  
+ {  
+   "cell_type": "markdown",  
+   "metadata": {},  
+   "source": [  
+     "Hi!"  
+   ]  
}
```

Nbdiff

- View changes made to notebooks
 - With the `diff` shell command

```
$ diff -c old.ipynb new.ipynb
```

- With the `nbdime nbdiff` command

```
$ nbdiff old.ipynb new.ipynb
```

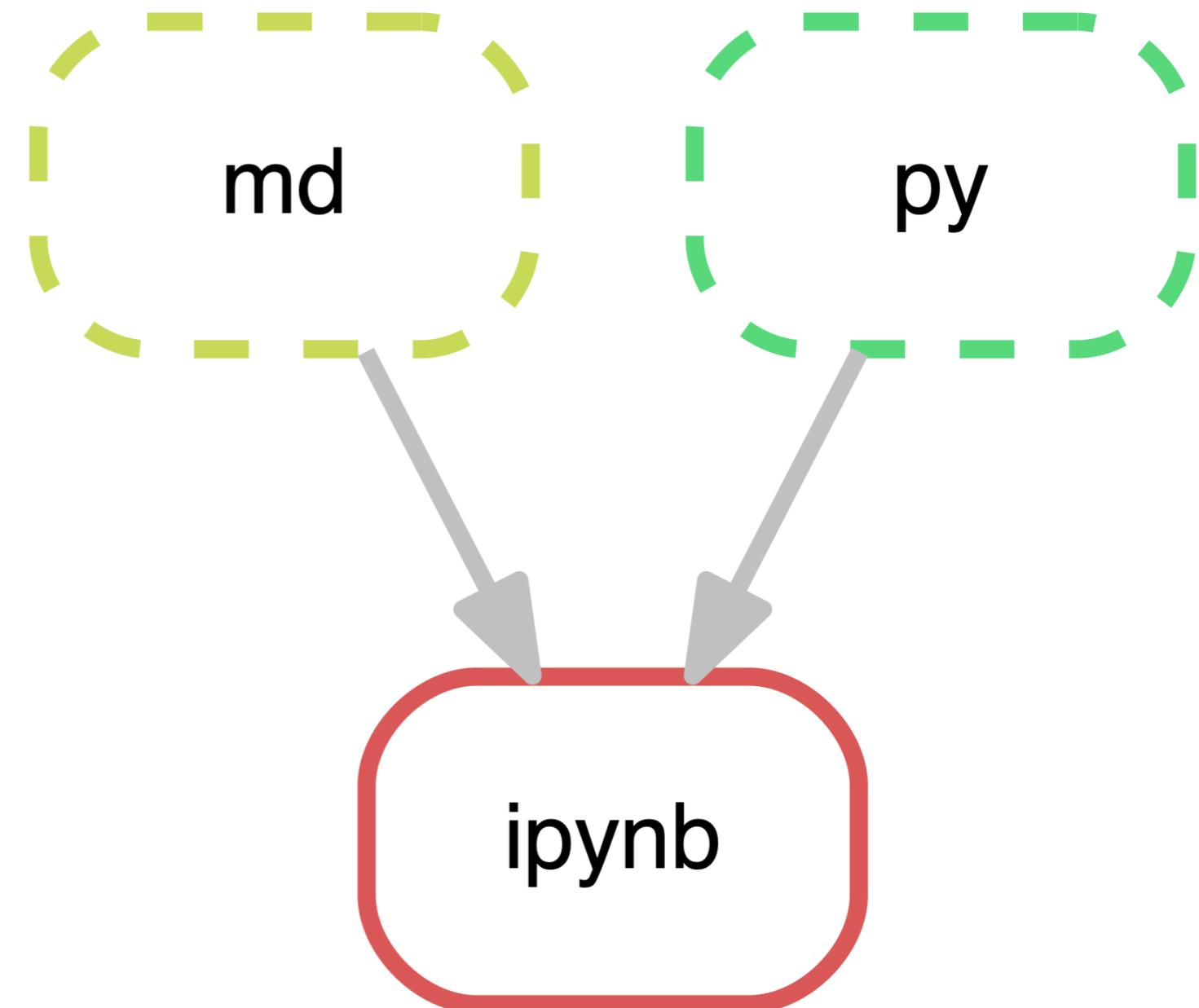
- <https://nbdime.readthedocs.io>

```
--- old.ipynb 2020-02-07 20:46:26.4981
+++ new.ipynb 2020-02-07 20:41:18.5494
## inserted before /cells/1:
+ markdown cell:
+   source:
+     Hi!
```

Notebook workflow package

1. Use `nbformat` to create notebooks from:

- Markdown files (`.md`)
- Code files (`.py`)

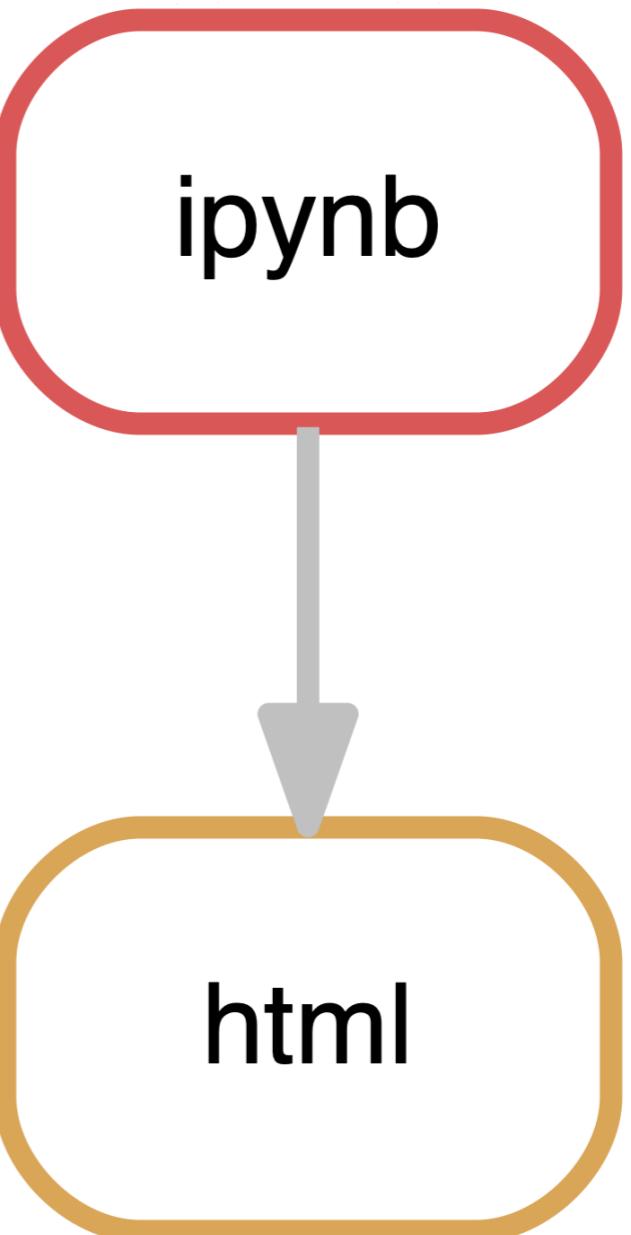


Convert notebooks

1. Use `nbformat` to create notebooks from:

- Markdown files (`.md`)
- Code files (`.py`)

2. Use `nbconvert` to convert notebooks



Code cells

```
from nbformat.v4 import (new_notebook,  
                         new_code_cell)  
  
nb = new_notebook()  
  
nb.cells.append(new_code_cell('1+1'))  
  
nb.cells
```

```
[{'cell_type': 'code', 'metadata': {},  
 'execution_count': None,  
 'source': '1+1', 'outputs': []}]
```

- Use `nbformat`'s `v4` module to create:
- Notebook objects
 - `new_notebook()`
- Code cell objects
 - `new_code_cell()`
- Code cell keys
 - `execution_count`
 - `source`
 - `outputs`

Unexecuted code cells

From Altair's [Example Gallery](#):

```
[ ]: import altair as alt
      from vega_datasets import data
      alt.Chart(data.cars()).mark_circle(size=60).encode(
          x='Horsepower', y='Miles_per_Gallon', color='Origin',
          tooltip=['Name', 'Origin', 'Horsepower', 'Miles_per_Gallon']
      ).interactive()
```

- Square brackets ([]:) on the left
 - Correspond to execution_count key-value pair

Executed code cells

From Altair's [Example Gallery](#):

```
[1]: import altair as alt
from vega_datasets import data
alt.Chart(data.cars()).mark_circle(size=60).encode(
    x='Horsepower', y='Miles_per_Gallon', color='Origin',
    tooltip=['Name', 'Origin', 'Horsepower', 'Miles_per_Gallon']
).interactive()
```



Running notebook code cells

- Increments the
 - Number in []: (rendered)
 - `execution_count` value
- Produces output (e.g. a plot)
 - Below the code cell
 - In the `outputs` list

Markdown cells

```
from nbformat.v4 import (
    new_markdown_cell
)
nb.cells.append(new_markdown_cell('Hi'))
len(nb.cells)
nb.cells[1]
```

```
2
[{'cell_type': 'markdown',
 'source': 'Hi', 'metadata': {}}]
```

- Use `nbformat`'s `v4` module to create:
- Markdown cell objects
 - `new_markdown_cell()`

```
import nbformat
nbformat.write(nb, "mynotebook.ipynb")
```

Nbconvert exporters

- Convert notebooks
 - Import and instantiate exporter

```
from nbconvert.exporters import HTMLExporter  
html_exporter = HTMLExporter()
```

- Obtain via by `get_exporter()`

```
from nbconvert.exporters import get_exporter  
html_exporter = get_exporter('html')()
```

Export files

- Create an HTML report from a Jupyter notebook:
 - Pass a notebook filename to the exporter's `from_filename()` method

```
contents = html_exporter.from_filename('mynotebook.ipynb')[0]
```

- Save the `contents` of the converted file

```
from pathlib import Path
```

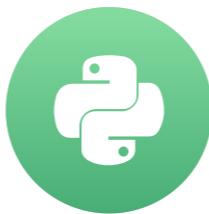
```
Path('myreport.html').write_text(contents)
```

Let's write a package for Jupyter notebook workflows!

CREATING ROBUST PYTHON WORKFLOWS

Pytest

CREATING ROBUST PYTHON WORKFLOWS



Martin Skarzynski

Co-Chair, Foundation for Advanced
Education in the Sciences (FAES)

Pytest tests

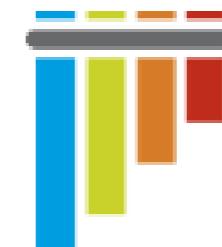
```
import pytest

def test_addition():
    assert 1 + 2 == 3

@pytest.mark.parametrize('n', [0, 2, 4])
def test_even(n):
    assert n % 2 == 0

def test_assert():
    with pytest.raises(AssertionError):
        assert 1 + 2 == 4
```

- Are functions
- Typically use `assert` statements
- Can test
 - Multiple values (`@parametrize`)
 - For expected errors (`raises()`)



pytest

Test-driven development

```
def double(n: float) -> float:  
    """Multiply a number by 2."""
```

```
from double import double  
  
def test_double():  
    assert double(2) == 4
```

- Define a function
 - With a docstring, but no code block
- Write a test
- Run the failing test

```
$ pytest test_double.py
```

Run failing tests

```
===== FAILURES =====
----- test_double -----
def test_double():
>     assert double(2) == 4
E     assert None == 4
E     +  where None = double(2)

test_double:4: AssertionError
===== 1 failed in 0.10 seconds =====
```

- Write a function
 - With a docstring, but no code block
- Write a test
- Run the failing test

```
$ pytest test_double.py
```

- Work on the module until it passes

```
def double(n: float) -> float:
    """Multiply a number by 2."""
    return n * 2
```

Run passing tests

```
===== test session starts =====  
platform linux -- Python 3.7.2, ...  
rootdir: PROJECT_PATH, configparser: ...  
plugins: mypy-0.3.2  
collected 1 item  
  
test_double . [100%]  
  
===== 1 passed in 0.03 seconds =====
```

- Write a function
 - With a docstring, but no code block
- Write a test
- Run the failing test

```
$ pytest test_double.py
```

- Work on the module until it passes

```
def double(n: float) -> float:  
    """Multiply a number by 2."""  
    return n * 2
```

Test change

```
===== FAILURES =====
----- test.raises -----
def test_raises():
    with pytest.raises(TypeError):
        double('2')
E       Failed: DID NOT RAISE
E       <class 'TypeError'>

test.raises.py:6: Failed
== 1 failed, 1 passed in 1.6 seconds ==
```

```
import pytest
from double import double

def test_raises():
    with pytest.raises(TypeError):
        double('2')

$ pytest test_raises.py
```

Make change

```
===== test session starts =====  
platform linux -- Python 3.7.2, ...  
rootdir: PROJECT_PATH, configparser: ...  
plugins: mypy-0.3.2  
collected 1 item  
  
test.raises . [100%]  
  
===== 2 passed in 0.32 seconds =====
```

```
def double(n: float) -> float:  
    """Multiply a number by 2."""  
    if type(n) == float:  
        return n * 2  
    else:  
        raise TypeError
```

```
$ pytest test_raises()
```

Make change

```
===== test session starts =====  
platform linux -- Python 3.7.2, ...  
rootdir: PROJECT_PATH, configparser: ...  
plugins: mypy-0.3.2  
collected 1 item  
  
test.raises . [100%]  
  
===== 2 passed in 0.32 seconds =====
```

```
def double(n: float) -> float:  
    """Multiply a number by 2."""  
    return n * 2.  
  
$ pytest test.raises()
```

Project organization

```
myproject
?
?? tests
?  ?? test_mymodule.py
?  ?? test_raises.py
?
?? src
?? mypackage
    ?? __init__.py
    ?? double.py
```

- Test files (e.g. `test_mymodule.py`)
 - Kept in `tests/` directory
- Modules (e.g. `double.py`)
 - Kept in packages
 - Along with `__init__.py`



Nbless

A Python package for
programmatic Jupyter
notebook workflows



5

Navigation

Contents:

[Project overview](#)

[Command-line interface](#)

[Module reference](#)

[Test reference](#)

Quick search

 Go

Welcome to Nbless's documentation!

Contents:

- [Project overview](#)
 - [Installation](#)
 - [Basic usage: terminal](#)
 - [Basic usage: Python environment](#)
 - [Too many file names to type out?](#)
- [Command-line interface](#)
 - [nbless](#)
 - [nbuild](#)
 - [nbconv](#)
 - [nbexec](#)
- [Module reference](#)
 - [nbless package](#)
- [Test reference](#)
 - [test_nbless module](#)

Indices and tables

- [Index](#)
- [Module Index](#)
- [Search Page](#)

Fork me on GitHub

Nbless

A Python package for
programmatic Jupyter
notebook workflows



5

Navigation

Contents:

[Project overview](#)

[Command-line interface](#)

[Module reference](#)

▪ [nbless package](#)

▪ [Subpackages](#)

▪ [nbless.helpers package](#)

▪ [nbless.main package](#)

▪ [Module contents](#)

[Test reference](#)

Quick search

 Go

- [nbless.main package](#)

- [Submodules](#)

- [nbless.main.nbconv module](#)

- [nbless.main.nbexec module](#)

- [nbless.main.nbless module](#)

- [nbless.main.nbread module](#)

- [nbless.main.nbsave module](#)

- [nbless.main.nbuild module](#)

- [Module contents](#)

Module contents

`nbless.nbuild(filenames: List[str]) → nbformat.notebooknode.NotebookNode`

[\[source\]](#)

Create an unexecuted Jupyter notebook from markdown and code files.

Parameters: `filenames` – A list of source file names.

`nbless.nbexec(nb_name: str, kernel: str = 'python3') → Tuple[str,`

`nbformat.notebooknode.NotebookNode]`

[\[source\]](#)

Create an executed notebook without modifying the input notebook.

Parameters: • `nb_name` – The `NotebookNode` object to be executed.

• `kernel` – The programming language used to execute the notebook.

`nbless.nbconv(nb_name: str, exporter: str = 'python') → Tuple[str, str]`

[\[source\]](#)

Convert a notebook into various formats using `nbformat` exporters.

Parameters: • `input_name` – The name of the input notebook.

• `exporter` – The exporter that determines the output file type.

Note: The exporter type must be ‘asciidoc’, ‘pdf’, ‘html’, ‘latex’, ‘markdown’, ‘python’, ‘rst’, ‘script’, or ‘slides’. pdf requires latex, ‘notebook’ does nothing, slides need to be served (not self-contained).

Let's practice writing tests!

CREATING ROBUST PYTHON WORKFLOWS