

Oscar Arbelaez Echeverri, Conrad Johnston

Converting Python 2 to Python 3

Writing compliant hybrid code, the AiiDA way



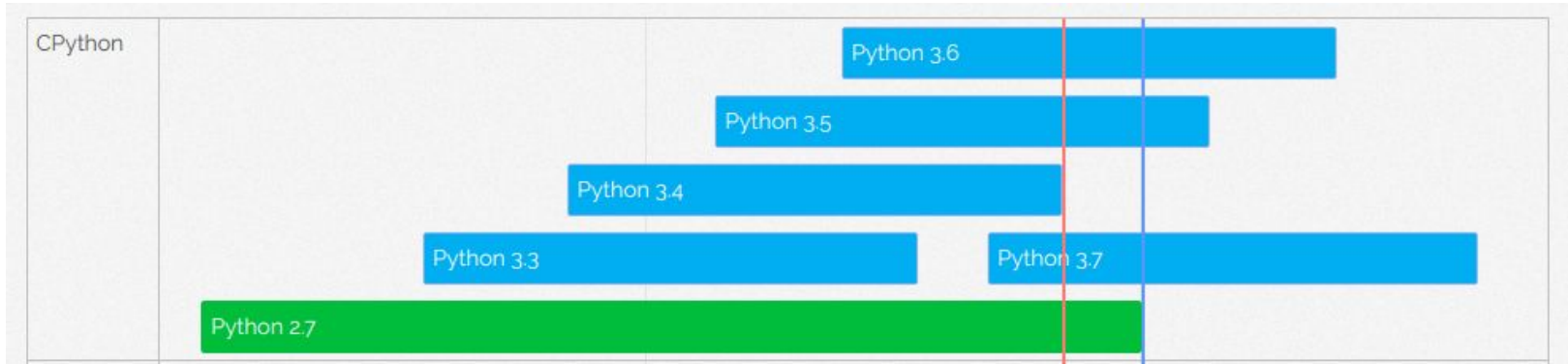
- AiiDA changes 0.x to 1.0
- Python 3 support

- This covers python 2 to 3
- We want the migration to be
painless

Motivation

Python 2 is at its lifetime end

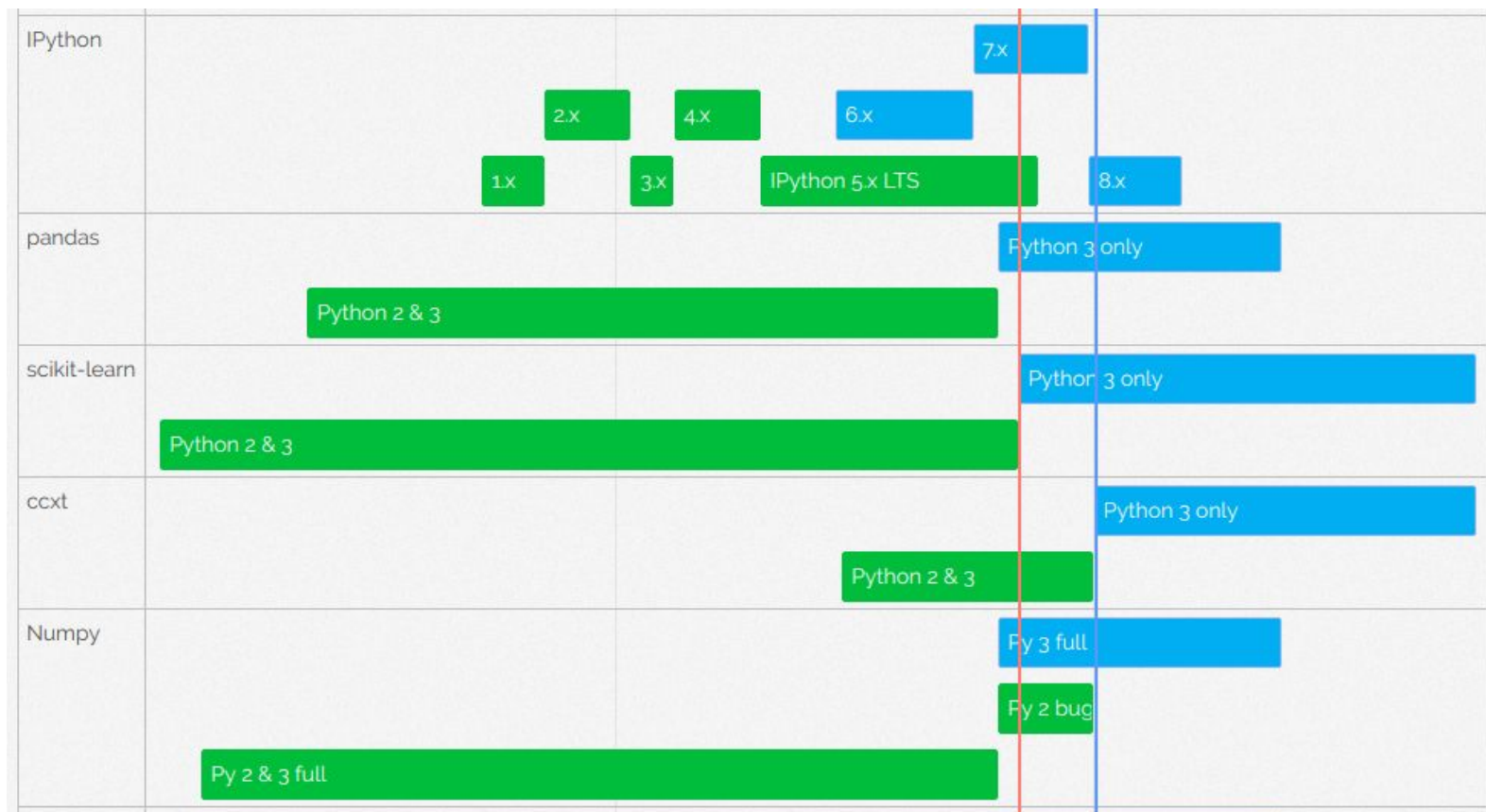
<https://python3statement.org/>





What does that mean?

- Your python2 installation is not going away
 - In a way you can still use it, and it will probably live forever (in our hearts)
- No new features will be added to python2
 - If you find a bug or a problem with python2 you're alone, people is not going to fix it
 - New versions of libraries will not support python2 (you'll see why)



Libraries will be or have already dropped support for python2 <https://python3statement.org/>

**Python 3.7 is also the best
python yet™**

```
In [1]: result = 1.5
```

```
In [2]: print(f'Our result is {result}')
```

Our result is 1.5

```
In [3]: print(f'Half result is {result / 2}')
```

Half result is 0.75

F-strings are awesome (python 3.6+)

```
In [1]: import numpy as np
```

```
In [2]: a = np.array([[1, 0], [0, 1]])
```

```
In [3]: b = np.array([[4, 1], [2, 2]])
```

```
In [4]: a @ b
```

```
Out[4]:
```

```
array([[4, 1],  
       [2, 2]])
```

They added a matrix multiplication operator for us scientists

```
In [1]: ñame = "la mejor fruta"
```

```
In [2]: simulación = 123
```

```
In [3]: print(ñame)
```

la mejor fruta

```
In [1]: command = 'new variable C hi there'
```

```
In [2]: instruction, *arguments = command.split(maxsplit=3)
```

```
In [3]: instruction
```

```
Out[3]: 'new'
```

```
In [4]: arguments
```

```
Out[4]: ['variable', 'C', 'hi there']
```

Unpacking is really handy



You can have all of that and...

- Safer comparisons
- Better memory management
- Better exception handling
- Keyword only arguments
- And more...

However...

Python 3 is not yet supported in all the platforms you might want to run on, looking at you clusters.

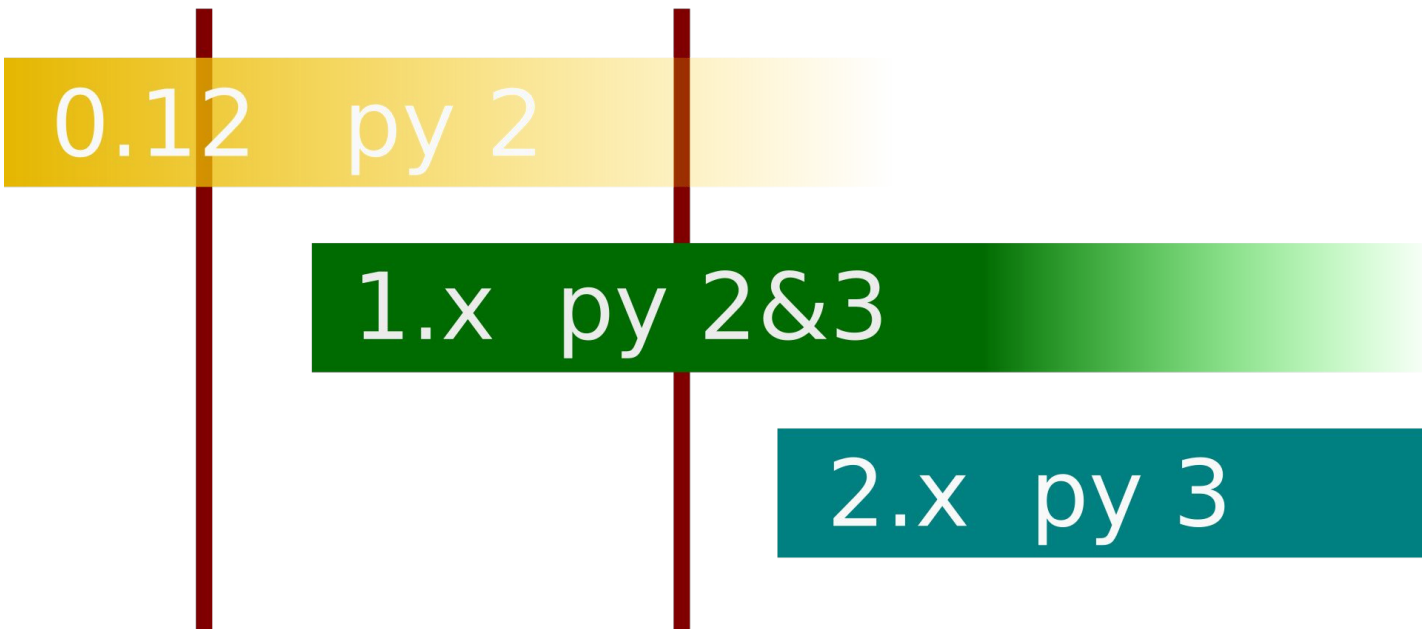
So for a while there's a need to support both python 2 and python 3



Photo credit OCLF, licenced as CC BY, <https://www.flickr.com/photos/olcf/>

Today

2020



(* we don't have exact dates yet) AiiDA is also dropping support for python 2

AiiDA 1.x py 2	Plugin py 2	Plugin py 2&3	Plugin py 3
AiiDA 1.x py 3	Plugin py 2	Plugin py 2&3	Plugin py 3
AiiDA 2.x	Plugin py 2	Plugin py 2&3	Plugin py 3

Be compatible with all the AiiDA versions we just talked about by supporting py 2 and 3 (* maybe 2.x)

- Update your plugin code to run in python 3.6 and above
- Please support python 2 & 3

**Let some tools migrate the code
for you**



lib2to3

- Python's own
- Would convert python 2 code into python 3 code

<https://docs.python.org/3.6/library/2to3.html>

```
def greet(name):  
    print "Hello, {0}!".format(name)  
print "What's your name?"  
name = raw_input()  
greet(name)
```

```
python -m lib2to3 python2.py -w
```

```
def greet(name):  
    print("Hello, {0}!".format(name))  
print("What's your name?")  
name = input()  
greet(name)
```



six

- Runtime dependency
- Helps you know if you're working on a python 2 or python 3 environment
- Provides “wrappers” so that your code works consistently between python 2 and 3

<https://six.readthedocs.io/>

```
import six

if six.PY2:
    # do something special for py2
    pass

data = {'a': 1, 'b': 2}
for key in six.iterkeys(data):
    six.print_(key)
```



python-future

- Makes python 2 compatible with both python 2 and 3
- Makes python 3 code compatible with python 2 and 3

<https://python-future.org/>

```
def greet(name):  
    print "Hello, {0}!".format(name)  
print "What's your name?"  
name = raw_input()  
greet(name)
```

futurize python2.py -w

```
from __future__ import print_function  
from builtins import input  
def greet(name):  
    print("Hello, {0}!".format(name))  
print("What's your name?")  
name = input()  
greet(name)
```



python-future

- Makes python 2 compatible with both python 2 and 3
- Makes python 3 code compatible with python 2 and 3

<https://python-future.org/>

```
def greet(name):  
    print("Hello, {0}!".format(name))  
print("What's your name?")  
name = input()  
greet(name)
```

`pasteurize python2.py -w`

```
from __future__ import print_function  
# ...  
from builtins import input  
from future import standard_library  
standard_library.install_aliases()  
def greet(name):  
    print("Hello, {0}!".format(name))  
# ...
```



python-modernize

- Makes python 2 compatible with both python 2 and 3
- Uses six as runtime dependency

<https://python-modernize.readthedocs.io/en/latest/>

```
def greet(name):  
    print "Hello, {0}!".format(name)  
print "What's your name?"  
name = raw_input()  
greet(name)
```

`python-modernize python2.py -w`

```
from __future__ import print_function  
from six.moves import input  
def greet(name):  
    print("Hello, {0}!".format(name))  
print("What's your name?")  
name = input()  
greet(name)
```



Other tools we recommend

Used in AiiDA-Core development:

- [YAPF](#) - automatically formats your python code
- [Pylint](#) - finds some issues with your code before you run it
- [Prospector](#) - analyses your python code
- [Pre commit](#) - runs your code “fixers” or any other command on commit

Another nice tool:

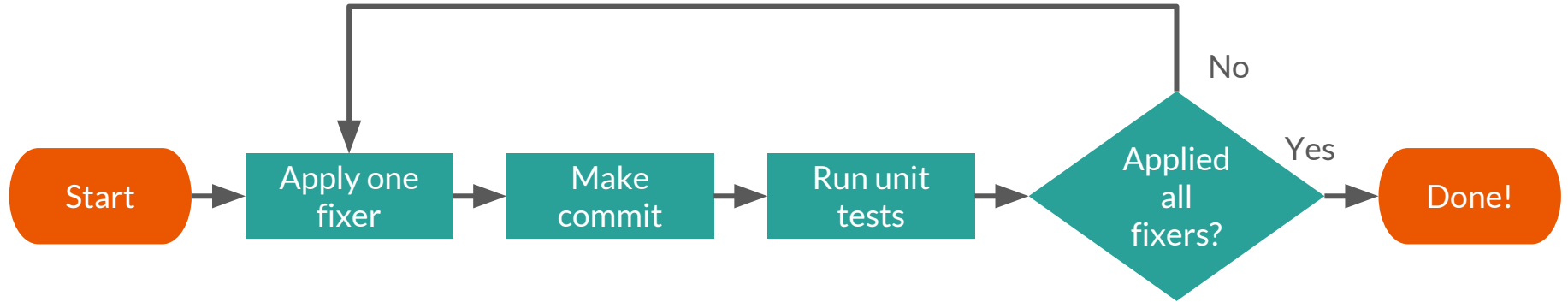
- [Black](#) - no questions asked formatting of your python code

We are happy to support you if you decide to use one of these.



How-to

Porting Workflow



Python Modernize Example : 2to3 fixers

Python 2

```
def metric_to_imperial(meters):  
    """Convert meters to feet and inches"""  
    total_inches = meters / 0.0254  
    feet = int(total_inches) / 12  
    inches = total_inches - (feet * 12)  
    print "%s m is %s ft %.3f in"\  
        %(meters, feet, inches)
```

Python 2/3

```
from __future__ import print_function  
def metric_to_imperial(meters):  
    """Convert meters to feet and inches"""  
    total_inches = meters / 0.0254  
    feet = int(total_inches) / 12  
    inches = total_inches - (feet * 12)  
    print("%s m is %s ft %.3f in"\  
        %(meters, feet, inches))
```

From running:

```
python-modernize -w example.py
```

Python Modernize Example : 2to3 fixers

Python 2

```
def metric_to_imperial(meters):  
    """Convert meters to feet and inches"""  
    total_inches = meters / 0.0254  
    feet = int(total_inches) / 12  
    inches = total_inches - (feet * 12)  
    print "%s m is %s ft %.3f in"\  
        %(meters, feet, inches)
```

Python 2/3

```
from __future__ import division  
from __future__ import print_function  
def metric_to_imperial(meters):  
    """Convert meters to feet and inches"""  
    total_inches = meters // 0.0254  
    feet = int(total_inches) // 12  
    inches = total_inches - (feet * 12)  
    print("%s m is %s ft %.3f in"\  
        %(meters, feet, inches))
```

From running:

```
python-modernize -f default -f classic_division example.py -w
```



Python Modernize Example : Six

Python 2

```
def conversion_calc():  
    input_length = raw_input(  
        "Enter a length in meters:"  
    )  
    metric_to_imperial(float(input_length))
```

Python 2/3

```
from six.moves import input  
def conversion_calc():  
    input_length = input(  
        "Enter a length in meters:"  
    )  
    metric_to_imperial(float(input_length))
```

From running:

```
python-modernize -f default -w
```



Details



Dictionaries

	Python 2	Python 3
keys() / values() / items()	Returns list	Returns view
has_key()	Works	Use in operator instead
Key ordering	“Not” ordered	< 3.6 : Not ordered 3.6 : Insertion ordered (implementation) >= 3.7 : Insertion ordered (language feature)



Dictionaries

```
# Python 2
```

```
d = {'a': 1, 'b' : 2, 'c': 3}
```

```
keys = d.keys()
values = d.values()
items = d.items()
```

```
if d.has_key('a'):
    print "Found 'a'"
```

```
if keys != ['a', 'b', 'c', 'd']:
    print "Missing key!"
```

```
# Python 2/3
```

```
from __future__ import print_function
d = {'a': 1, 'b' : 2, 'c': 3}
```

```
keys = d.keys()
values = d.values()
items = d.items()
```

```
if 'a' in d.keys():
    print("Found 'a'")
```

```
if sorted(keys) != ['a', 'b', 'c', 'd']:
    print("Missing key!")
```

Dictionaries: Views

Views are dynamic, not static.

```
# Python 2

d = {'a': 1, 'b' : 2}

keys = d.keys()

print keys[0]

d['c'] = 3

if 'c' in keys:
    print "Found key: 'c'"
```

```
# Python 3

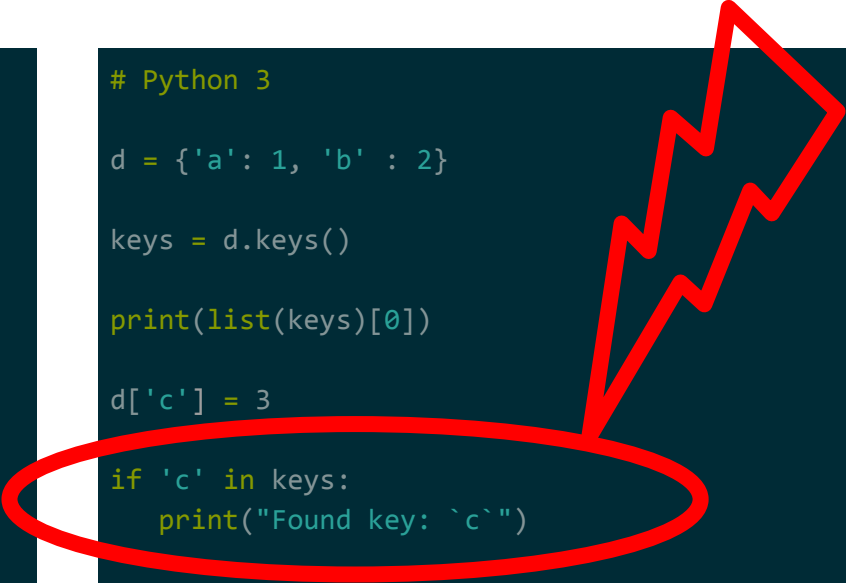
d = {'a': 1, 'b' : 2}

keys = d.keys()

print(list(keys)[0])

d['c'] = 3

if 'c' in keys:
    print("Found key: `c`")
```

A red lightning bolt is drawn on the right side of the Python 3 code block, pointing towards the difference in key lookup. A red oval is drawn around the 'if' statement in the Python 3 code block, highlighting the use of backticks for string literals.



Dictionaries: Iterators

No `iteritems()` in Python 3, but `items()` returns a view which can be iterated

```
# Python 2
```

```
d = {'a': 1, 'b' : 2, 'c': 3}

for key, value in d.iteritems():
    print key, value
```

```
# Python 2/3
```

```
from __future__ import print_function

d = {'a': 1, 'b' : 2, 'c': 3}

for key, value in d.items():
    print(key, value)

import six
for key, value in six.iteritems(d):
    print(key, value)
```



Exceptions

```
# Python 2

def do_something(x):
    raise ValueError('Value of x must spark joy')

try:
    do_something(1)
except Exception as exc:
    log = exc.message
    print "Bad value: {}".format(exc.message)

try:
    do_something(1)
except Exception as exc:
    raise BonusException(exc)
```

```
# Python 2/3
from __future__ import print_function

def do_something(x):
    raise ValueError('Value of x must spark joy')

try:
    do_something(1)
except Exception as exc:
    log = str(exc)
    print("Bad value: {}".format(exc.args[0]))

try:
    do_something(1)
except Exception as exc:
    raise six.raise_from(BonusException, exc)
```



Unicode

- ASCII: 7 bits, maps all unaccented English characters, numbers, punctuation
- ANSI: 8 bits, consisting of ASCII + **localisation dependent** bonus characters (code pages)
 - Can't mix code pages!
- DBCS: Up to two bytes, depending on character, for Asian languages
- Unicode: 16 bit (in principle, but not really)
 - Maps using `code points`
 - Example **U+0041**, maps to **A**
 - Encoding specifies how to store the code point in memory
 - UTF-8, UTF-16, UTF-32, ASCII, etc
 - UTF-x guarantees that the code point can be stored

Unicode vs Str vs Bytes

Python 2

```
some_string = "Important stuff"
type(some_string)    # <type 'str'>

some_unicode = u"\U0001f634"
type(some_unicode)    # <type 'unicode'>

print some_unicode
```



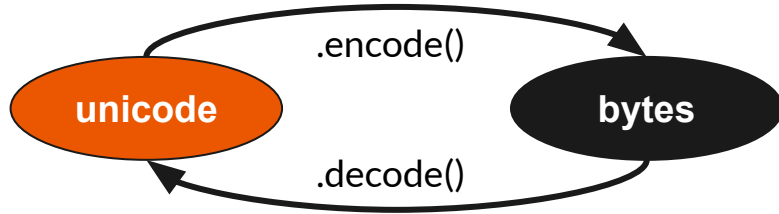
Python 3

```
some_string = "Important stuff \U0001f634"
type(some_string)    # <class 'str'>

some_bytes = b"Important stuff"
type(some_bytes)     # <class 'bytes'>

print(some_bytes)
```

.encode(), .decode()



```
# Python 2
some_unicode = u"\U0001f634"
len(some_unicode) # 1

bytes = some_unicode.encode('utf-8')
print bytes
len(bytes) # 4

print u"hello " + "there" # Implicit!
```

```
# Python 3

print "hello " + b"there" # TypeError!
```



File Handling

- Python 2 string are bytes, Python 3 string are Unicode
- Am I reading binary or text? If text, what is its encoding?

```
# Python 2/3

# For text files
import io
with io.open('myfile.dat', encoding='utf-8', mode='w') as fhandle:
    fhandle.write(s)

# For bytes files
with io.open('myfile.dat', mode='wb') as fhandle:
    fhandle.write(s)
```



Unicode Goals

1. Make a Unicode sandwich
 - Bytes on the outside, Unicode filling
2. Know what you have
 - Unicode or bytes?
 - For bytes, what encoding?

Summary

1. Python 2 support is ending imminently
2. AiiDA plugins need to support both 2 and 3
3. We want this to be painless for you!



Acknowledgments

Tiziano Müller, University of Zurich

EPFL

THEOS
THEORY AND SIMULATION
OF MATERIALS

MARVEL

NATIONAL CENTRE OF COMPETENCE IN RESEARCH

MAX DRIVING
THE EXASCALE
TRANSITION