
Fython

Release 1.0 alpha

Aug 24, 2017

Contents

1	Features	3
1.1	Performance	3
1.2	Python Syntax	3
1.3	Python Comfort	4
1.4	Pycessor	4
1.5	Template	5
2	Installation	7
3	Contribute	9
4	Support	11
5	License	13
5.1	Overview	13
5.2	Installation	13
5.3	Getting Started	14
5.4	Language Reference	27

Fython is Fortran with a Python syntax. If performance requirements periodically forces you out of Python, with Fython you won't feel it

hello.fy

```
real offset = 0

def fast_sum:
    real in v(1e9)
    real res r

    r = 0
    for i in [1, 1e9]:
        r += v[i] + offset

    print 'The sum is {:sum(v)}'
```

The loop above is automatically parallelized by the Fortran compiler that powers Fython.

Usage in Python is as simple as

hello.py

```
import fython
import numpy as np

hello = fython.load('.hello')

offset = hello.offset()
v = fython.Real(shape=[1e9])

offset = 10
v[:] = np.random.uniform(0, 1, 1e9)

hello.fast_sum(v)
```


CHAPTER 1

Features

Performance

The Fython language is a syntactic binding for the Fortran language. Any instruction you write in Fython is translated to Fortran, with the exact same meaning. The translated Fortran code is then compiled to machine instructions. There is no boiler plate code generation or LLVM in between. What you code is what you run (WYCIWYR).

Python Syntax

Fython allows to write Fortran code with Python syntax. No messy `end xyz` everywhere. No more `%` for object attributes, enjoy the `car.color` notation. No more concatenation puzzle, use multiline string `"""xyz"""`. No more `&` for multiline instruction, simply use parenthesis

```
many(  
    arguments,  
    and,  
    more,  
)
```

Write class with class, enjoying attribute getter and setter.

```
class Nucleon(Atom):  
    real position = 1  
    int weight = 10  
  
    def pget fission:  
        self inout  
        self.weight /= 2  
  
Nucleon n  
n.fission  
print 'weight {:n.weight}'
```

Python Comfort

Fython import system is modeled after Python. You can define Fython modules accross several folders. They will all correctly compile and link.

```
import .variance = var
import stat.mean(*)
import .svm(vapnik, least_square_svm=lsq_svm)
```

Fython has a stack trace for fast error spotting. Say goodbye to those uninformative runtime errors.

```
fython sigsegv: segmentation fault

module stat.buggy
function subboom
line 7

stack trace (lineno function module) (most recent first)

7 subboom    stat.buggy
3 boom       stat.buggy
```

Fython print and read function are intuitive. The format mini-language is that of Fortran plus several improvements. Interpolating variables can be specified directly in the string like Perl.

```
print './out' 'x is {:x}'

print .file 'column 2 is {v:x[:,2]}
```

```
read .data: x y z
```

Pycessor

Use Python to easily do your preprocessing magic

```
import numpy

real pi = |numpy.pi|

# lazy initialization of exotic random variate
real v(1e3)

|
  for i in range(1000):
    write('v[i] = {:f}'.format(numpy.random.uniform()))
|
```

Any expression enclosed in bars is evaluated against the imported Python module. The return value of a pycession can be any valid Fython code.

Template

Overload of a function or a class can be created with template interpolation

```
def temp f:
    T in x
    T res r
    r = x + 10

def g = f(T=real)
```

When this is not sufficient, a whole package can be templated

quicksort.fy

```
import type_provider(target_class=T)

def quicksort(x):
    T x(:)
    int i
    int res r

    r = 0
    for i in [1, size(x)]:
        r += x[i].less_than(x[i+1])
```

consumer.fy

```
import quicksort(*)
||type_provider = maxwell, target_class = Atom ||

int r
Atom a(10)
r = quicksort(a)
```

FyTypes

You can send Python object by reference to Fython using the intuitive fytypes

```
from fython import *

m = load('stat.mean')

nb_element = Int(value=3)
x = Real(value=[1, 2, 3])
result = Real()

m.mean(nb_element, x, result)

print(result[:])
```

When using a fytype, you always access or modify its value with a slice. Whether the value is a scalar or an array.

```
x[:2] += 10

result[:] *= 10
```

Changes made by Fython are propagated back to Python

```
m.moving_average_in_place(x)

print('moving average is:', x[:])
```

You can also access and modify global Fython variables

```
n = m.roundoff_tolerance()
n[:] = 1e-5
```

All the above works for real, integer and string variables of any dimension

```
cities = Char(size=30, value=['montreal', 'tokyo', 'santiago'])
result = Char(size=30)

m.word_mean(nb_element, cities, result)
```

Wrap Fortran code and shared library

All intrinsic Fortran module are available in Fython. Other Fortran modules are available once they are in your Python path

```
import iso_c_binding(*)
import fftw.discrete_cosine_transform(*)

real x(10)
fftw_dct(x)
```

Putting a Fortran module in the Python path is usually done with

```
import sys
sys.path.append('~/.fftw')
```

A Shared library can be imported once you have a Fortran or a Fython interface for it

```
import mkl.include.mkl_vsl(*)
import mkl.lib.intel64.libmkl_intel_lp64(*)
```

The first import is the fortran interface `mkl_vsl.f90`. The second import is for the shared object library `libmkl_intel_lp64.so`.

The requirement for these imports to work is that the mkl root directory must be in your Python path. This is usually achieved with

```
import sys
sys.path.append('/opt/intel')
```

CHAPTER 2

Installation

```
git clone https://github.com/nicolasessisbreton/fython
cd fython
python3 setup.py install
```

Dependencies:

- Any Fortran compiler
- Python 3
- PLY
- Tabulate

Platform:

Linux

CHAPTER 3

Contribute

Any contribution is welcomed. The source is available on [github](#).

CHAPTER 4

Support

Questions, suggestions, improvements or bug fixes? Join the discussion [here](#).

Fython is released under an [Apache License 2.0](#).

Overview

Installation

Fython can be installed with

```
git clone https://github.com/nicolasassisbreton/fython
cd fython
python3 setup.py install
```

To use Fython, a Fortran compiler must be present on your system. By default, Fython will try to use gfortran or ifort. If you want to explicitly specify one of these you can do

```
import fython

fython.use_ifort()
fython.use_gfortran()
```

To use another compiler do

```
import fython

fython.set_compiler(
    cmd = 'gfortran'
    prefix = '__',
    infix = '_MOD_',
    suffix = '',
    debug = '-O0',
    release = '-O3',
```

```
error_regex = '(Error:|ld:)'
)
```

Assuming

```
nm -D gfortran_produced.so
```

show names as

```
__modname_MOD_varname
```

that is the pattern

```
prefix modname infix varname suffix
```

Testing it

That's it. Your Fython rocket is ready

```
>>> import fython
>>> fython.hello()
'Welcome to Fython. See the build products in ./__fycache__'
```

Syntax Highlighting in Sublime

To get syntax highlighting in Sublime Text, you can use these [files](#).

Getting Started

Any Fython package must be part of a Python package. This way Fython can leverage Python automatic package detection system.

In this tutorial, we will create a Fython package `stat` that illustrates most of Fython Functionalities.

The code for this tutorial is in [fython/example/stat](#).

The Host Python package

First, we create the folder structure for the Python package where our Fython package will reside

```
stat/
  setup.py
  stat/
    mean/
      mean.fy
      mean_test.py
    variance/
      variance.fy
```

```

    variance_spec.fy
    variance_test.py

    newspaper/
    print.fy
    print_test.py
    read.fy
    read_test.fy

    class/
    class.py
    class_test.fy

    imports/
    __init__.fy
    imports.fy
    imports_test.fy

    pycessor/
    pycessor.fy
    pycessor_test.py

    baygon/
    baygon.fy
    baygon_test.py
    force_test.py
    release_test.py
    verbose_test.py

    backus/
    fortran.py
    so.py

```

language shell

The content of `setup.py` is

```

from setuptools import setup, find_packages

setup(
    packages = find_packages(),
    name = 'stat',
)

```

language python

We register the package in Python with

```
python3 setup.py develop
```

Mean

The Mean module illustrate the basic semantic of Fython. We write three kind of mean functions.

In Fython, the arguments of a function must have the intent attribute `in`, `out` or `inout`.

mean.fy

```

real cons boltzman = 10 # a constant

int plank = 8 # a global variable

def cube_mean:
    real in: x y z
    real out r

    r = x + y + z
    r /= 3
    r **= 3

    ##
    Yes, Fython has all the augmented assignment operators (+=, -=, *=, /=, ↪
    ↪ **=) .
    Yes, this is a multiline comment.
    ##

def moving_mean:
    int in n
    real inout x(n)
    int i

    # let's forget about the edges
    for i in [2, n-1]:
        x[i] = sum( x[i-1:i+1] ) / 3

    if x[1] > 5:
        print 'x[1] is bigger than 5: x[1]={:x[1]}'
        print ""
        All the values in x are:
        {v:x}
        ""

    ##
    The Python vector x will be modified in-place.

    The print format mini-language is that of Python, plus a few additions.
    The v directive eases the printing of vector.

    ##

def string_mean:
    char(3) in x(3)
    real out r
    int: i j

    # a mean on strings? creative.
    for i in [1, 3]:
        for j in [1, 3]:
            r += ichar( x[i][j:j] )

    r /= 10

    r += boltzman + plank # essential constants for any calculation, aren't ↪
    ↪ they?

    ##

```

The ichar function gives the ascii code of a character.

The ichar function **is** an intrinsic Fortran function.

Yes, you have access to **all** of them.

##

Testing Mean

Let's test drive our mean module

mean.py

```
from fython import *

m = load('.mean')

# accessing a global variable
p = m.plank()

# setting a global variable
p[:] = 6

# cube mean
print('cube mean')
x = Real(value=1)
y = Real(value=2)
z = Real(value=3)
r = Real()

m.cube_mean(x, y, z, r)

print(' result in python', r[:])

# moving mean
print('\nmoving mean 1')

n = Int(value=5)
x = Real(value=[1, 2, 3, 4, 5])
m.moving_mean(n, x)

print(' result in python', x[:])

# an other time
print('\nmoving mean 2')
x[0] = 10
x[1] = 20
m.moving_mean(n, x)

print(' result in python', x[:])

# string mean
x = Char(size=3, value=['abc', 'xyz', 'ijk'])
m.string_mean(x, r)

print('string mean', r[:])
```

language python

We did not attempt to access the constant `bolztman`. This is because variable defined with `as constant` are not accessible from fython.

That's it for the mean module. Now you can write a standalone Fython module. In preparation for those days where you need more, let's see the variance module.

Variance

In Fython the specification and the implementation of a function or a class can be separated. Similar to Python, we start by working on our implementation of the variance function, deferring the spec to the `variance_spec` import

variance.fy

```
import .variance_spec (*)

def variance:
    cube_mean(x, y, z, u)
    r = x - u + maxwell # this is the official formula on planet kawabunga
```

When we are satisfied with our algorithm, we write the specification

variance_spec.fy

```
import stat.mean.mean (*)

def variance:
    real in: x y z
    real out r
    real u
    int cons maxwell = 8
```

The specification contains all the imports and the definitions we need in `variance.fy`. We test with

variance_test.py

```
from fython import *

m = load('.variance')

# cube mean
print('calling variance')
x = Real(value=1)
y = Real(value=2)
z = Real(value=3)
r = Real()

m.variance(x, y, z, r)

print(' result in python', r[:])
```

What we did above is called an implicit spec interpolation. We can also do explicit spec interpolation with the `spec` modifier

explicit_spec_interpolation.fy

```
def f:
```

```

real in x
real out y
y = x + 2

def spec(f) g:
    y = x * 10

```

We test with

explicit_spec_interpolation_test.fy

```

from fython import *

m = load('.explicit_spec_interpolation')

print('calling f')
x = Real(value=1)
y = Real()
m.f(x, y)

print(' result in python', y[:])

print('calling g')
x = Real(value=1)
y = Real()
m.g(x, y)

print(' result in python', y[:])

```

Newspaper

Any good bug is remove by several usage of the print statement. With Fython the print statement output can be standard out, a file on the Python path, or a path on the file system

print.fy

```

int i
real x(10)

for i in [1, 10]:
    x[i] = i
    print 'x({:i}) = {:x[i]}'

print .x_final 'x is {v:x}'

# string to specify the path
print './x_transformed.out' """
    x+10 is: {vc:x+10}
    x-10 is : {vc:x-10}
    """

# Yes, Python multiline string

```

The Python url `.x_final` tells Fython to create a file ‘x_final.out’ in the same directory than the Fython module. A string can also be used to specify the path.

print_test.fy

```
import os
from fython import *

os.system('rm *.out') # cleaning any previous run

m = load('.print', force=1)

print('\nfinal x')
print(open('./x_final.out', 'r').read())

print('\nx transformed')
print(open('./x_transformed.out', 'r').read())
```

Since all bugs originates from data, we use the read statement to keep ourselves busy

read.fy

```
int: x y z
int u(3)

print .data mode(w) '1, 2, 3' # explicitly creating a new file with the mode_
↪modifier

read .data: x y z

print 'x {:x}'
print 'y {:y}'
print 'z {:z}'

# vectors too
read .data u
print 'u is {v:u}'
```

The possible mode for printing to files are mode (a) for appending, and mode (w) for overwriting. The default mode is appending. We test with

read_test.py

```
from fython import *

m = load('.read')
```

Class

Similar to Python, the first argument to any class method must be `self`. You can use any name for the self argument, the only rule is that the first argument is the self argument.

class.fy

```
class Magnetism:
    real maxwell = 8
    real pointer tesla(:) => null()
    real allocatable bohr(:)

    def energy:
        self in # first argument is always self
        real res r
```



```

    r = self.maxwell + sum(self.tesla)

def pget courant:
    self in
    real res r
    r = self.maxwell + sum(self.bohr)

def pset courant:
    s inout # any name is allow for the self argument
    real in value
    s.bohr = value

real target x(10) = 1
Magnetism m

m.tesla => x # pointer assignment
print 'energy {:m.energy()}'

# happy allocation
alloc m.bohr(8)
print 'bohr {v:m.bohr}'

# getter/setter
m.courant = 4.
print 'courant {:m.courant}'

# happy deallocation
dealloc m.bohr

```

Inheritance and spec interpolation are also possible. See the language section for more details.

class_test.py

```

from fython import *

m = load('.class')

```

Imports

Three kinds of imports are possible in Fython. With a star import all the names are imported. With an aliased namespace import, the names of the target module are accessible through the alias. With a slice import, only the stated names are imported, possibly aliased.

imports.fy

```

import ..mean.mean (*)

import ..variance.variance = v

import ..newspaper.print (
    i = counter, # aliasing name i
    x, # no alias
)

import ..imports = package_import # when the directory as a __init__.fy file

print 'mean boltzman {:boltzman}'

```

```
counter = 1
v.variance(1., 2., 3., x[counter])
print 'variance {:x[counter]}'

# triggering package main code
package_import.main()
```

imports_test.py

```
from fython import *

load('.imports')
```

It is possible to import a directory when it contains a `__init__.fy` file. This is usefull as your package grow. The content of our `__init__.fy` for the imports directory is

`__init__.fy`

```
import iso_c_binding(*)
print 'package export'
```

When importing a package, the main code of the package needs to be triggered manually. The main code of package is any code that is not part of a function and that is not a specification (variable, class and interface specification).

Pycessor

A Pycession is the fythonic term for Python preprocessor interpolation.

Pycessions can be used to define compile time constant or to avoid writing lines of code that are similar.

Since any Python code is allowed in pycession, a more exotic usage can be to run a Makefile script that produce a shared library used in your module.

For clarity, we write python imports together with the other fython imports at the top of a module.

pycessor.fy

```
import os
import numpy
# or any python module on your path

real pi = |numpy.pi| # atan what? no more

# lazy initialization of exotic random variate
real v(1e3)

|
x = numpy.random.uniform(0, 1, 1000)

for i in range(1000):
    write('v[i] = {:f}'.format(x[i]))
|

# running a make file
|os.system("echo 'compiling boost ... done'")| # for real? isn't it 2 hours
```

pycessor_test.py

```
from fython import *

load('.pycessor')
```

When a pycession is an expression such as `1+2` or `f(1)`, its returned value is inserted in your fython code. When a pycession contains several lines, you need to explicitly state wich string to include in your code. The special pycessor function `write` serves this purpose.

Baygon

When an error occurs in your code, Fython will usually detect it and produce a stack trace.

baygon.fy

```
def boom:
    real x
    subboom(x)

def subboom:
    real inout x
    x = 1 / 0
```

baygon_test.py

```
from fython import *

m = load('.baygon')

# shocking hazard
m.boom()
```

If Fython error detection system is overridden by your compiler or simply fails, you can use the verbose function of a Fython module, The verbose function tells Fython to print the location of every line of code that are run. You can then easily spot that wonderfull bug.

verbose_test.py

```
from fython import *

m = load('.baygon')

m.verbose()

m.boom()
```

Sometimes Fython may fails to detect changes in your code since the last compilation. If that happens, simply load your module with the `force` option to trigger a refresh of the Fython cache

force_test.py

```
from fython import *

m = load('.baygon', force=1)

m.boom()
```

When your code is bug free, the `release` keyword tells Fython to run your code at full Fortran speed, with all optimizations enables

release_test.py

```
from fython import *

m = load('.baygon', release=1)

m.boom()

print('a bad idea here as the division by zero will go unseen')
```

Template

Function template are usefull to create overload of a function

function.fy

```
def temp f:
    T in x
    print 'x is {:x}'

def f_real = f(T=real)
def f_int = f(T=int)

f_real(1.5)
f_int(1)
```

function_test.py

```
from fython import *
load('.function')
```

The template function needs to me marked with the modifier `temp`. The principle is the same for class

class.fy

```
class temp Atom:
    T x

    def lt:
        self in
        self in other
        bool res r
        r = self.x > other.x

def Electron = Atom(T = real)

Electron e
Electron p

e.x = 10
p.x = 1

print '{:e.lt(p)}'
```

class_test.py

```
from fython import *
load('.class')
```

You can also templatize a whole package

package.fy

```
import .class(*)
import .quicksort(*)
||type_provider=.class, target_class=Electron||

Electron e(10)
int i

for i in [10, 1, -1]:
    e.x = i

quicksort(e)
```

When the module quicksort and all of its dependency is imported any occurrence of `type_provider` and `target_class` will be replaced by the package interpolation provided at the import statement.

The content of quicksort is

quicksort.fy

```
import asis type_provider(target_class=T)

def quicksort:
    T dimension(10) in x
    int: i r
    for i in [1, 9]:
        r = x[i].lt( x[i+1] )
        print 'i {:r}'
```

To prevent any package interpolation to happen during the import of `type_provider`, the import has the `asis` modifier.

We test with

package_test.py

```
from fython import *
load('.package')
```

Backus

You can import a Fortran module in Fython. For this example, we use the `writer` function of Fython.

fortran.py

```
from fython import *

writer("""
.brent.f90
```

```
module brent
  integer, parameter :: tolerance = 1e-3

contains

  function root(x) result(r)
    real, dimension(10) :: x
    real :: r
    integer :: i

    r = 0

    do i = 1, 10
      r = r + x(i)
    end do

  end function

end module

.consummer.fy

import .brent(*)

real x(10) = 1

print 'brent says {:root(x)}'
""")

load('.consummer')
```

The `writer` function turns a Python script into a playground for languages. The function creates the file `.brent.f90` in the current directory. The content of the file is the indented content after the file name.

What we did is that we created the file `brent.f90` and the file `consummer.fy`. The Fython `consummer` module imports the Fortran `brent` module. We then test the Fython `consummer` module with the `load` function.

we can also imports shared library in Fython

so.py

```
import os
from fython import *

writer("""
.ritchie.f90
  module ritchie
    real, bind(c) :: c = 10

  contains

    subroutine compile() bind(c)
      write(*, *) 'c is ', c
    end subroutine

  end module

.backus.fy
```

```

import .ritchie(*)

c = 20

compile()
"""

os.system('gfortran ritchie.f90 -shared -fpic -o ritchie.so')

load('.backus')

```

In `ritchie`, we use the `bind(c)` attribute to emulate the standard naming convention in shared library. We then compile `ritchie` into a shared library with `gfortran`. After that we load the Fython module `backus` into Python. The `backus` module then places a class to `compile`.

Language Reference

This section describes in details the Fython language.

In code examples

- shell code is indicated with a dollar `$`
- Python code is indicated with triple quote `>>>`
- Fortran code is indicated with triple bangs `!!!`

When no mention of the language is made, assume it is Fython.

The term shared library is often simply referenced by `so`.

Syntax

In Fython a statement is formed by a keyword, modifiers and target

```
keyword modifier* target
```

The keyword is the action performed by the statement. The modifiers mutate the default behavior of the actions, The action dictated by the keyword and modifiers is then applied to all the target.

```

real pointer x

real pointer: x y z
real pointer: x, y, z

real pointer:
  x
  y
  x

real pointer:
  x, y, z
  a, b, c

```

The comma is necessary for target that are non atomic

```
real cons: x=1, y=2, z=3
```

The only exception to the statement construction are in-place assignment operation

```
real: x y  
  
x = y + 1  
x += 10
```

Modifiers are also called attributes.

Any modifier that is allowed in Fortran can also be used in Fython

```
real pointer x  
int allocatable contiguous y
```

Arrays are defined by indicating their dimensions

```
real:  
  x(10)  
  y(1:10, 0:5)
```

Array elements are accessed with the slice notation

```
x[1:6] => y[2, :]
```

You can initialize an array or use an array in place with the bracket notation

```
real x(3) = [1, 2, 3]  
  
f([1, 2, 3])
```

Imports

Three kinds of imports are possible in Fython. Aliased namespace import, star import and slice import.

```
import pca  
import stat.mean = m  
  
import stat.variance(*)  
import stat.newspaper(x, y=z)
```

When the module url is composed of only one name, such as the `pca` imports. The statement is equivalent to

```
import pca = pca
```

With an aliased namespace import, the object in the module are access with a dot .

```
import stat.mean = m  
m.cube_mean(1, 2, 3)
```

With a star import all the object of the imported module are available

```
import stat.mean(*)  
cube_mean(1, 2, 3)
```


With a slice import only the stated names are imported, optionally aliased

```
import stat.mean(cube_mean, char_mean= cm)
cube_mean(1, 2, 3)
char_mean('abc', 'def', 'ijk')
```

For all imports it is necessary that you have write permission to the directory that contains the imported module. This is because Fython needs to maintain build products in the same directory. The only exception to this rule are shared library import, as no build product needs to be maintained in these case.

The url of a module is its qualified Python name

```
import numpy.random.uniform
```

This implies that for a file to be importable, it must be on your Python path.

The first way to put a file on your Python is to create a host Python package and registering it to Python with a setup script

```
$ python3 setup.py develop
```

See the Getting Started section for the details.

The other method is to modify your path directly

```
#>>>
import sys
import fython
sys.path.append('/opt/nag')
m = fython.load('random.sobolev')
```

In a Python url, the file extension cannot be part of the url. You should then take into account the following resolution order. In a compile time import, Fython first search for

- a Fython file (.fy, __init__.fy)
- a Fortran file (.f90, .f03, ... and many other)
- a So File (.so)

In a pycessor time import, Fython search for

- a Python file (.py, __init__.py).

For print and read statement that uses an url, the assumed extension is .out.

For Fortran, only star and slice imports are allowed. For So only star imports are allowed.

Usefull modifiers of the import statements are

```
import asis payoff_defs_provider(*)
import noforce mkl.include.mkl_vsl(*)
```

The `asis` modifier prevents any package interpolation to happen during the import. This is usefull when designing a packaged meant to be a template. See the Template section.

The `noforce` modifier prevents a forcefull recompilation. If the module already exists, it is not recompile, even if it was loaded with `load(url, force=1)`. This is usefull to avoid recompilation of heavy module, that anyway never changes.

Declaration

The declarations order follows Fortran convention. Variables, Classes and interface declaration should appear first within a scoping unit, then functions.

A scoping unit is the entire module, the body of a class, or the body of a function.

For this release, nested class or nested function are not supported.

Operator

Fython has the augmented assignment operators, the logical operator, the bitwise operators, and the pointer operator.

```
x += 1
x -= 1
x *= 2
x /= 2

x <<= 1
x &= 1
x ^= 1
x |= 1
x >>= 1

x < <= == != => > y # this is an invalid syntax

x and y or b not c

x >> 1 + y << 4

x => y # pointer
```

The min and max operator are often convenient.

```
x += y # x = max(x, y)
x -= y # x = min(x, y)
```

Variable Declaration

In Fython the elementary types have a Python flavor

```
real x
int y
char z
bool a
complex b
```

Constant are declared with the attribute `cons`.

```
real cons x
```

Classes are instantiated by using there name

```
class A:
    pass

A a
```

String variable can be assign a value with a multiline string

```
char(100) x

x = """
    extra leading space
    at the beggining remove
    """

x = '''
    triple quote
    '''

x = 'single line'
x = "double quote"
```

Any newline or tab character in the string will be honored.

For procedure argument, use the `proc` modifier

Coarray

A coarray is defined by specifying its codimension in bracket

```
real x[*]
real y(10)[*]
```

A coarray is accessed with the slice notation

```
x = 1 # this_image() x
x[2] = 2 # x on image 2

y[1] = 1 # this_image() y
y[:,2] = 1 # y on image 2
```

To use coarray in Fyhton, you need to set the compiler to use with the `set_compiler` function

```
#>>>
from fyhton import *

set_compiler(
    cmd = 'ifort',
    prefix = '',
    infix = '_mp_',
    suffix = '_',
    debug = '-coarray -coarray-num-images=5',
    release = '-coarray -coarray-num-images=5',
    error_regex = 'error:',
)

m = load('.coarray_test')
```

Function

A function is declared with the keyword `def`

```
def f:
    real in x
    real res r
```

For a variable to be recognized as an argument, it must have one of the intent modifier

```
in
out
inout
```

The return value of a function must be indicated with the modifier

```
res
```

When no argument has the modifier `res`, the function has no return value.

You can separate the implementation and the specification of you function with `spec` interpolation

```
def pure f:
    real in x
    real res r

def f:
    r = x + 3
```

The spec for `f` can be in the same module or originate from an import. You can also explicitly specify the spec to use with the spec.

```
def elemental f:
    real in x
    real res r

def spec(f) g:
    r = x + 10

    def spec(f, g) h:
        pass # multiple spec parent
```

You can use the `inline` instruction to include verbatim the definition of one function into another

```
def f:
    x += 1

def g:
    inline f
```

The modifier `debug` or `release` can be use to specify in which mode to include the code. This is usefull for conditional inclusion of logging code for example.

```
inline debug f
inline release f
```

When no modifier is specified, the code is inlined in all compilation mode.

Automatic argument completion dispense for the need to write all the arguments of a function provided the name of the argument is the same than a name in the current scope.

```
real x = 1
real y = 10

def f:
    real in x
    real in y

f(y=1.) # x added automatically
f() # both x and y added
```

Automatic arguments completion works for keyword arguments call only. It cannot be mixed with positional argument code.

```
# with f as above

f(y) # not supported
f(y=1.) # supported
```

If a function should not be compiled, used the `noproduct` modifier. This is usefull when the function is only used as a spec provider, and that the function should not be compiled.

```
def noproduct f:
    real x(n) # n is not defined, this would give an error if compiled

def spec(f) g:
    int in n

    x += 1 # definition of x is provided by the spec of f
```

The `noproduct` modifier is not inherited during a spec interpolation. So, only `f` is not compiled. To not compile `g`, explicitly use the modifier `noproduct`.

to help distinguish between pure and non-pure function used the modifiers `pure` and `sidef`

```
def pure f:
    pass

def sidef g:
    pass
```

The modifier `sidef` has no effect during compilation. The modifier clearly states the intent of the coder: that the function `g` has side-effects, and cannot be marked as pure.

Interface

An interface is declared with the `interface` keyword

```
interface:
    def f:
        real in x
        real res r
```

To facilitate the definition of C procedure the modifier `iso(c)` can be used

```
interface:
  def iso(c) f:
    real in x
```

The `iso(c)` modifier can be used on any function declaration and is not restricted to interface declaration. The effect of the modifier is to produce

```
!!!
subroutine f(x) bind(c)
  use iso_c_binding
  real, intent(in) :: x
end subroutine
```

Class

A class is defined with the `class` keyword.

```
class A:
  real x

  def f:
    self in
    real res r
    r = self.x

  def pget y:
    s in
    real res r
    r = s.x

  def pset y:
    s inout
    real in value
    s.x = value
```

The first argument of any class method must be the `self` argument. The name used for the `self` argument can be anything. Above we used `s` instead of `self` for the getter and setter.

Getter and Setter are defined with the `pget` and `pset` modifiers.

Inheritance is indicated with parenthesis after the class name

```
class C(B, A):
  pass
```

You can separate the specification and the implementation of a class with the `spec` interpolation

```
# spec.fy

class A:
  real x
  def pure f:
    self in
    real res r

# code.fy
import spec(*)
```

```
class A:
    def f:
        r = self.x + 10
```

You can explicitly state the spec to use with the spec modifier

```
class A:
    real x

class spec(A) B:
    pass
```

You can use the `inline` statement to include verbatim the definition of a class or a function into your class

```
class A:
    real x

class B:
    inline A
```

Allocation

Memory is allocated and deallocated with the `alloc` and `dealloc` keyword

```
alloc: x y(n) z

alloc(n):
    x
    y
    z(m)

dealloc: x y z
```

When the keyword `alloc` has an argument, it is used as the default size for any variable where no size is specified.

Control Flow

Fython has `if`, `for`, `fop`, `while` and `where` statement

```
if x < 1:
    y += 1

elif x < 1:
    y += 2

else:
    y = 0
```

The third argument in the bracket of a `for` statement is the step size

```
for i in [1, 2]:
    r += x[i]
```

```
for i in [0, 10, 2]:  
    r += x[i] # 0, 2, 4, ...
```

The `fop` loop is a parallel for loop. The Fortran equivalent is a `do concurrent` loop.

```
fop i in [1, 2]:  
    r += x[i]
```

The `while` loop is

```
while x < 10:  
    x += 1
```

The `where` statement is

```
where x > 1:  
    x = y  
  
elwhere x < 1:  
    x -= 1  
  
else:  
    x = 0
```

Print

Printing to the console needs no modifier

```
print 'x {:x}'
```

When an url is used the file extension is assumed to be `.out`

```
print .simulation 'x {:x}'
```

A file system path can also be used

```
print './simulation.out' 'x {:x}'
```

You can then choose any extension you want.

You can print to a character variable when its name does not contain a dot

```
char(100) r  
  
print r 'x {:x}'
```

If the name contains a dot use the `unit` modifier

```
print unit(atom.name) 'proton'
```

The `unit` modifier can also be used if you opened a file by yourself

```
int u = 10  
open(unit=u, file='./simulation.out')  
print unit(u) 'x {:x}'
```


If you use a number, the unit modifier is not necessary

```
print 10 'x {:x}'
```

You can control the mode in which the file is written to during a print statment with the mode modifier

```
print mode(a) 'x {:x}'
print mode(w) 'overwrite any previous content'
```

The default mode is a.

To continue printing on the same line, use the c modifier

```
print c 'start '
print c ' on same line'
print ' ending the line'
print 'this one on a new line'
```

The format mini-language is that of Fortran plus several additions

```
print """
{:x} : general format used
{f5.2:x} : float format
{i5:x} : int format

{v:y} : vector format: [1, 2, 3, ]
{vc:y} : vector content format: 1, 2, 3,
{va:y} : vector format: array([1, 2, 3, ]) ; usefull for python post-processing
"""
```

The additions are the v, vc and va formats that facilitates the printing of vectors.

Format that helps printing to the JSON format are also available. The JSON formats avoid typing the name of a variable twice, and helps to deal with comma.

```
print """
{jn:x} : json no comma before: "x": x
{j:x} : json with comma before: ,"x":x
{jv:x} : json vector: "x":[1, 2, 3]
{jvn:x} : json vector no comma before: ,"x":[1,2,3]
{j_tag:x} : json with specified tag: ,"tag":x
{jv_tag:x} : json vector with specified tag: ,"tag":[1,2,3]
{jn_tag:x} : json no comma before with specified tag: "tag":x
{jvn_tag:x} : json no comma before vector with specified tag: "tag":[1,2,3]
"""
```

In a Typical printing with JSON format, the first element is explicitly specified without leading comma, then the remaining elements are added, prepended by a comma.

```
print """
[
  { } # first element no comma

  ,{ } # any addition prepended by a comma

  ,{
    {jn:x} # no comma
    {j:y} # prepended by a comma

  }

]
"""
```

If a print statement is used only in debug mode, use the `xip` instruction

```
xip 'printed only in debug mode'
print 'printed in both debug and release mode'
```

The `xip` takes the same modifiers than the `print` instruction. The `xip` instruction is usefull for debugging.

Read

You can read a file by specifying its url. The extension is then assumed to be `.out`

```
read .data: x y z
```

You can specify a path in the file system with a string

```
read './data.out': x y z
```

You are then free to use any extension you want.

The `read` statement in Fython supports csv-like formats automatically. In Fortran, this is a called a list-directed read. For this release, the other kind of read statement are not supported.

You can use the name of variable that does not contains a dot for the read source

```
char(100) data

read data: x y z
```

If the name of the variable contains a dot, use the `unit` modifier

```
read unit(mendelev.table): x y z
```

You can read into a vector or any other variable

```
read .data:
  x[:, i]
  atom.name
```

FyTypes

Only three kinds of data type can travel back and forth between Python and Fython

```
#>>>
from fython import *
x = Real()
y = Int()
z = Char(size=100)

m = load('.mean')

m.f(x, y, z)
```

Only function that have no return value can be called from Python. In Fortran term, `f` must be a subroutine.

Fython can modify in-place the element send by Python. The change will be seen in Python. The same is true in Python. In change made to a ftype in Python, will be seen in Fython.

The value of a ftype is always accesses with a slice, wheter the ftype is a scalar or a vector

```
#>>>
x = Real()
y = Real(value=[1, 2, 3])

x[:] = 9
y[:1] = 10 + x[:]
```

The three Fytypes all have the optional arguments `size`, `shape` and `value`. They are shown below with their default value

```
#>>>
x = Real(size=4, shape=[], value=None)
y = Int(size=4, shape=[], value=None)
z = Char(size=100, shape=[], value=None)
```

An empty list indicates a scalar. A vector is defined by specifying the number of element in each dimension.

```
#>>>
x = Real(shape=[10, 10])
```

The argument `value` is used to connect a ftype to a Python object. Any change made to the ftype will be reflected in the Python object

```
#>>>
x = [1, 2, 3]
y = Real(value=x)
```

To access a global variable, use its name

```
#>>>
from fython import *
m = load('.mean')
tolerance = m.tolerance()
```

Fython will automatically detects the type of the variable and use the default ftype initializer above. You can specify the variable specification yourself

```
#>>>
x = m.x(size=8, shape=[10])
```

Once setted the shape of fytype cannot change. This limitation can be overcome by letting Python and Fython share informations

```
#>>>
m = load('.mean')
m.compute()
n = m.result_size()
result = m.result(size=n[:])
```

Callback

In Fython, it is possible to call a Python function. Such callable function are called callback. The trick is to pass the address of the callback. This address is a simple integer, so, in fython, we cast it to a function pointer.

The code below shows how to transfer integer, real and arrays of these two.

The Fython code goes as follows.

```
import:
    iso_c_binding(*)

interface:
    def iso(c) py_fct:
        int(c_int) value in:
            xint
        real(c_float) value in:
            xreal
        int(c_int) value in:
            nintv
            nrealv
        int(c_int) in:
            xintv(*)
        real(c_float) in:
            xrealv(*)

def f:
    int(8) in:
        py_fct_pointer_int
    c_funptr:
        py_fct_pointer
    proc(py_fct) pointer:
        pyf
    int:
        x
        xv(2)
    real:
        y
        yv(2)

    print 'fython start: {:py_fct_pointer_int}'
    py_fct_pointer = transfer(py_fct_pointer_int, py_fct_pointer)
    c_f_procpointer(py_fct_pointer, pyf)
    print 'pyf called'
    x = 1
    y = 0.5
    xv = 10
    yv = 5.5
```

```
pyf(x, y, 2, 2, xv, yv)
print 'fython exit'
```

The Python goes like this

```
#>>>
from mttc import *
from ctypes import *
import numpy as np

m = load('.pycall', force = 2)

def f(xint, xreal, nintv, nrealv, xintv, xrealv):
    array_type = c_int*nintv
    addr = addressof(xintv.contents)
    xintv = np.array(array_type.from_address(addr), copy=0)

    array_type = c_float*nrealv
    addr = addressof(xrealv.contents)
    xrealv = np.array(array_type.from_address(addr), copy=0)

    print('hello from python', xint, xreal, nintv, nrealv, xintv, xrealv)

c_fun_dec = CFUNCTYPE(None, c_int, c_float, c_int, c_int, POINTER(c_int), POINTER(c_
↪float))

c_fun = c_fun_dec(f)

c_fun_int = cast(c_fun, c_void_p).value

m.f(
    py_fct_pointer_int = Int8(c_fun_int),
)
```

Load Function

The python api side load function has the following default arguments

```
#>>>
load(
    fython_module_url,
    force = 0,
    release = 0,
)
```

The force argument is used to force a refresh of the Fython dependency cache.

The release argument indicates wheter to run in debugging mode (0), or release mode (1). In debug mode, Fython tries to detect errors. In release mode, all the compiler optimization are enabled.

When a Fython module is loaded, you access its objects with there name.

```
#>>>
from fython import *
m = load('.mean')

x = m.global_variable()
```

```
y = m.global_variable_with_explicit_shape(shape=[10])

# function call
m.mean(x, y)

# keyword call
m.mean(
    offset = x,
    vec = y,
)
```

When a global variable is accessed, Fython will automatically determine its fitype. For function however, only minimal arguments consistency is made. You should make sure that you invoke your Fython function with the right argument order and the right fitype. The argument order consistency can be alleviated by using a keyword argument call. Fython will then call your Fython function with the argument in the right order.

Pycessor

Pycession instruction are specified within bars. The Python imports necessary for the pycession at the top of your Fython module

```
import numpy = np

pi = |np.pi|
```

A pycession can be multiline

```
|
  for T in ['real', 'int']:
    write('def f_{T:s} = g(T = {T:s})'.format(T))
|
```

The signature of the `write` function is

```
#>>>
write(
    string,
    *args,
    end = '\n',
    **kwargs,
)
```

The positional and keyword arguments are used to format the `string` argument.

When a pycession is a Python expression, its value is directly inserted in your Fython code. The `write` function is necessary when the pycession is not an expression. Each string passed to the `write` function is inserted in your Fython code. The `end` argument is appended to the string.

Any valid Python code is possible in a Pycession.

The Pycessor is a preprocessor. Do not use it to pass arguments to Fython because the dependency system will not see any post-compilation change in your Python module. The Pycessor is meant to facilitate the generation of tedious code, or to trigger any kind of necessary preparation before compilation.

Template

A function template is defined with the `def` statement

```
def f:
    T in x
    T res r
    r += x

def g = f(T=real)
```

To templatize a whole package use the import statement

```
import quicksort(*)
||type_provide=stat.mean, target_class=KMean||
```

Package interpolation can also be multilines

```
import quicksort(*)
||
    type_provide = stat.mean
    target_class = KMean
||
```

To disable package interpolation during a package import, use the `asis` modifier.

```
import asis random
```

This is usefull to avoid further interpolation during an ongoing package interpolation.