Programación Astronomica

Week 9

Horas de oficina

Miércoles y Jueves 9:00 - 10:00

o pueden comunicase conmigo por email

para pedir hora

o pueden pasarse directamente por mi oficina:

Oficina 210 de Astro.

Aviso: este Miércoles 10 de Oct. estaré fuera de la oficina entre las 9:00 - 13:00.

Week 9

- Data types
- Numpy
- Indización
- Archivos de ASCII: reading & writing simplified

Recuerden siempre con que “data types” están trabajando en su código,

y

comparen con que data types NECESITAN trabajar.

Eso es, necesitan entender si están usando floats, integers, strings, booleans, etc, y si el uso y contexto es correcto

Data types

```
x = 3
print(type(x)) # Prints "<class ‘int'>"

y = 2.5
print(type(y)) # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"

t = True
print(type(t)) # Prints "<class ‘bool'>

hello = 'hello'    # String literals can use single quotes
print(hello)       # Prints "hello"
```

https://cs231n.github.io/python-numpy-tutorial/

https://docs.python.org/3.5/library/stdtypes.html#string-methods

Lists

Una lista es el equivalent en Python de un arreglo, pero con taman~o adjustable y que suede container differences “data types”

```
xs = [3, 1, 2]    # Create a list
print(xs, xs[2])  # Prints "[3, 1, 2] 2"
print(xs[-1])     # Negative indices count from the end of the list;
prints "2"
xs[2] = 'foo'     # Lists can contain elements of different types
print(xs)         # Prints "[3, 1, 'foo']"
xs.append('bar')  # Add a new element to the end of the list
print(xs)         # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()      # Remove and return the last element of the list
print(x, xs)      # Prints "bar [3, 1, 'foo']"
```

https://docs.python.org/3.5/tutorial/datastructures.html#more-on-lists

Slicing

In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as *slicing*:

```
nums = list(range(5))  # range is a built-in function that creates a list of integers
print(nums)            # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])       # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
print(nums[2:])        # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])        # Get a slice from the start to index 2 (exclusive); prints "[0, 1]"
print(nums[:])         # Get a slice of the whole list; prints "[0, 1, 2, 3, 4]"
print(nums[:-1])       # Slice indices can be negative; prints "[0, 1, 2, 3]"
nums[2:4] = [8, 9]     # Assign a new sublist to a slice
print(nums)            # Prints "[0, 1, 8, 9, 4]"
```

Lists “comprehensions”

Consideren este código para computar cuadrados de un arreglo de numero enteros:

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)   # Prints [0, 1, 4, 9, 16]
```

Podemos simplificar sibgnitivamente usando “list comprehensions”:

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)   # Prints [0, 1, 4, 9, 16]
```

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)  # Prints "[0, 4, 16]"
```

Diccionarios

If you want access to keys and their corresponding values, use the `items` method:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

Dictionary comprehensions: These are similar to list comprehensions, but allow you to easily construct dictionaries. For example:

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)  # Prints "{0: 0, 2: 4, 4: 16}"
```

https://docs.python.org/3.5/library/stdtypes.html#dict

“Tuples”

Un “tuple” es una lista ordenada (inmutable) de valores. Un tuple es en muchos aspectos similar a una lista; una de las diferencias más importantes es que los tuples se pueden usar como claves/llaves en los diccionarios y como elementos de conjuntos, mientras que las listas no pueden. Aquí hay un ejemplo trivial:

```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
t = (5, 6)          # Create a tuple
print(type(t))      # Prints "<class 'tuple'>"
print(d[t])         # Prints "5"
print(d[(1, 2)])    # Prints "1"
```

https://docs.python.org/3.5/tutorial/datastructures.html#tuples-and-sequences

```
>>> d = {(x, x+1): x for x in range(10)}
>>> print(d)
{(0, 1): 0, (1, 2): 1, (6, 7): 6, (5, 6): 5, (7, 8): 7, (8, 9): 8, (4, 5): 4, (2, 3): 2, (9, 10): 9, (3, 4): 3}
>>> print(type(d))
<type 'dict'>
>>> print(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> t = (5,6)
>>> print(type(t))
<type 'tuple'>
>>> print(d[t])
5
>>> print(d[(t)])
5
>>> print(d[(5,6)])
5
>>> print(d[(1,2)])
1
>>> print(d[(0)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
>>> print(d[0])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
>>> print(d[(0,1)])
0
```

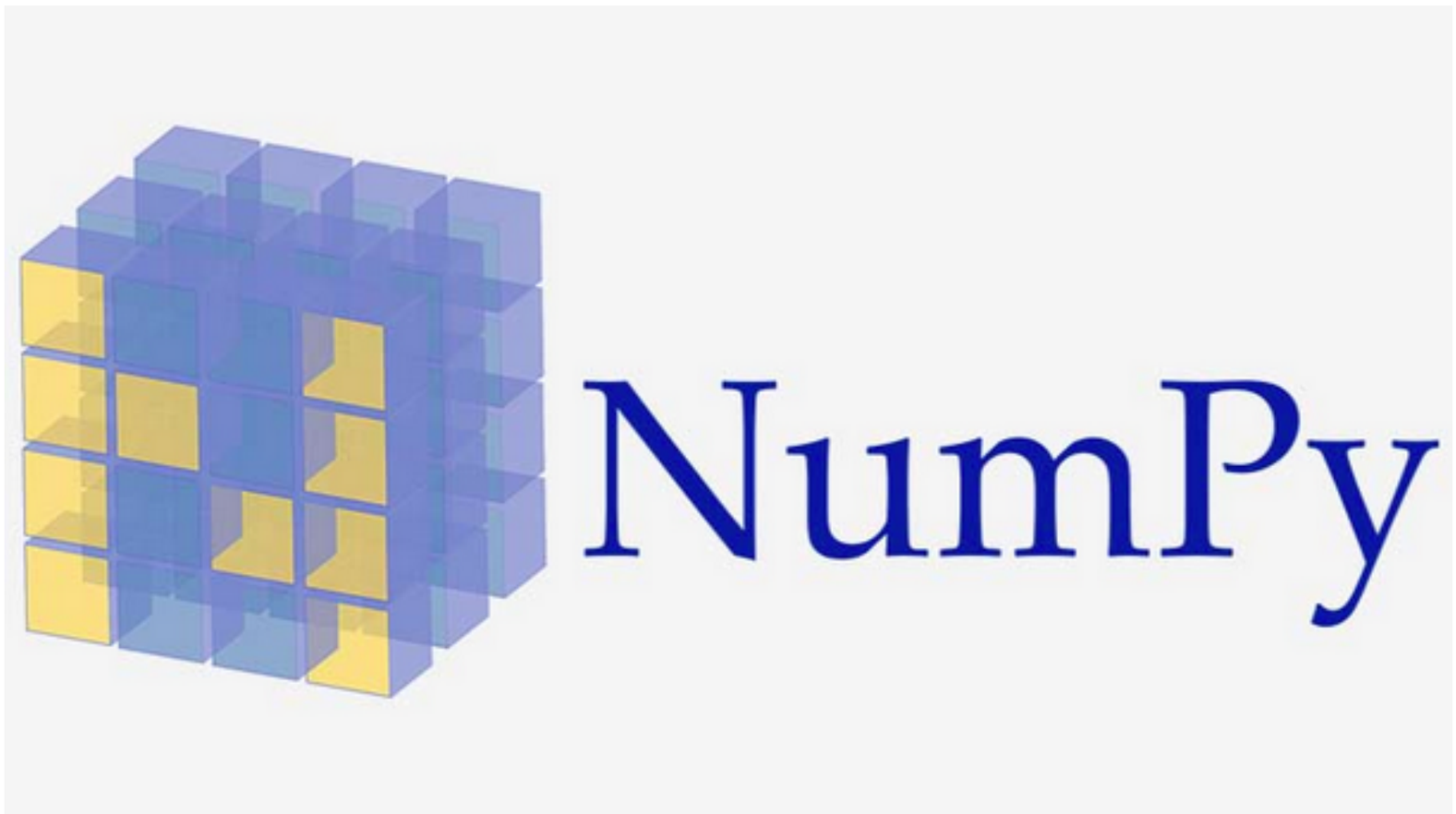
Funciones

https://docs.python.org/3.5/tutorial/controlflow.html#defining-functions

Clases: las veremos mas tarde.

https://docs.python.org/3.5/tutorial/classes.html

NUMPY: NUMerical Python



http://www.numpy.org/

NumPy es el paquete fundamental para la computación científica con Python. Contiene entre otras cosas:

- Un poderoso objeto de matriz N-dimensional
- Funciones sofisticadas (difusión)
- Herramientas para la integración de código C / C ++ y Fortran.
- Álgebra lineal útil, Fourier análisis, y capacidades de números aleatorios.

Además de sus obvios usos científicos, NumPy también se puede usar como un eficiente contenedor multidimensional de datos genéricos. Se pueden definir tipos de datos arbitrarios. Esto permite que NumPy se integre a la perfección con una amplia variedad de bases de datos.

Listas de Python:

- son muy flexibles
- no requieren tipos numéricos uniformes
- Son muy fáciles de modificar (insertar o anexar objetos).

Sin embargo, la flexibilidad a menudo conlleva el costo del rendimiento, y las listas no son el objeto ideal para los cálculos numéricos.

Aquí es donde entra **Numpy**. Numpy es un módulo de Python que define un poderoso objeto de matriz n-dimensional que utiliza C y Fortran detrás para proporcionar un alto rendimiento.

La desventaja de los arreglos de Numpy es que tienen una estructura más rígida y requieren un solo tipo numérico (por ejemplo, valores de punto flotante), **pero para muchos trabajos científicos, esto es exactamente lo que se necesita.**

```
import numpy as np
```

Creando NUMPYY arrays

```
a = np.array([10, 20, 30, 40])
a.ndim   #nos da el numero de dimensiones (1)
a.shape  #nos da la forma (4,)
a.dtype  #tipo numerico ('int64')
```

Nota: los arrays de Numpy en realidad admiten más de un tipo de entero y un tipo de punto flotante; admiten enteros de 8, 16, 32 y 64 bits con signo y sin signo, y flotantes de 16, 32 y 64 bits.

https://astrofrog.github.io/py4sci/_static/10.%20Introduction%20to%20Numpy.html

Una matriz numpy es una “cuadrícula” (“**grid**”) de valores, todos del mismo tipo, y está indexada por un tuple de números enteros no negativos.

El número de dimensiones es el rango (“**rank**”) de la matriz
La forma (“**shape**”) de una matriz es un tuple de enteros que da el tamaño de la matriz en cada dimensión.

Podemos inicializar matrices numpy desde listas anidadas de Python y acceder a elementos mediante corchetes:

https://cs231n.github.io/python-numpy-tutorial/

Creando NUMPYY arrays

```
import numpy as np

a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

Creando NUMPYY arrays

```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #              [ 0.68744134  0.87236687]]"
```

Creando NUMPYY arrays: la función `arange`

```
np.arange(10)

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.arange(1.2, 4.4, 0.1)

array([ 1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9,  2. ,  2.1,  2.2,
        2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9,  3. ,  3.1,  3.2,  3.3,
        3.4,  3.5,  3.6,  3.7,  3.8,  3.9,  4. ,  4.1,  4.2,  4.3])
```

Creando NUMPYY arrays: la función `linspace`

Otra función útil es el linspace, que se puede usar para crear valores espaciados linealmente entre los límites incluidos:

```
np.linspace(11., 12., 11)

array([ 11. ,  11.1,  11.2,  11.3,  11.4,  11.5,  11.6,  11.7,  11.8,
        11.9,  12. ])
```

Creando NUMPYY arrays: la función `logspace`

```
np.logspace(1., 4., 7)

array([    10.        ,     31.6227766 ,    100.        ,    316.22776602,
       1000.        ,   3162.27766017,  10000.        ])
```

Los arrays de Numpy se pueden combinar numéricamente usando los operadores estándar + - * / **

```
x = np.array([1,2,3])
y = np.array([4,5,6])
```

```
x + 2 * y
```

```
array([ 9, 12, 15])
```

```
x ** y
```

```
array([  1,  32, 729])
```

Esto es DIFERENTE al comportamiento de listas:

```
x = [1,2,3]
y = [4,5,6]
```

```
x + 2 * y
```

```
[1, 2, 3, 4, 5, 6, 4, 5, 6]
```

```
import numpy as np


x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

* es la multiplicación “elementwise” (de elemento a elemento), no la multiplicación de matrices. En su lugar, utilizamos la función “de punto” (dot product) para calcular los productos internos de los vectores, para multiplicar un vector por una matriz, y para multiplicar las matrices. `dot` está disponible como una función en el módulo numpy y como un método de instancia de objetos de matriz:

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])
v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

Transposing a matrix.

```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #          [3 4]]"
print(x.T)  # Prints "[[1 3]
            #          [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)    # Prints "[1 2 3]"
print(v.T)  # Prints "[1 2 3]"
```

Data types

Cada matriz numpy es una cuadrícula de elementos del mismo tipo. Numpy proporciona un gran conjunto de tipos de datos numéricos que puede utilizar para construir matrices. Numpy intenta adivinar un tipo de datos cuando crea una matriz, pero las funciones que construyen matrices generalmente también incluyen un argumento opcional para especificar explícitamente el tipo de datos.

```
import numpy as np

x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

https://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html

Acceso y corte de arreglos
Accessing and Slicing arrays

Similar a listas, se puede acceder a elementos de arreglos via indices:

```
x = np.array([9,8,7])
x[0]
9
x[1]
8
```

and arrays can also be **sliced** by specifying the start and end of the slice **(where the last element is exclusive)**:

```
y = np.arange(10)
y[0:5]
array([0, 1, 2, 3, 4])
```

optionally specifying a step:

```
y[0:10:2]
array([0, 2, 4, 6, 8])
```

Arreglos multi-dimensionales

```
x = np.array([[1.,2.],[3.,4.]])
x.ndim
2
x.shape
(2, 2)
```

```
y = np.ones([3,2,3])  # ones takes the shape of the array, not the values
array([[[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]],

       [[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]],

       [[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]]])

y.shape
(3, 2, 3)
```

Slicing multi-dimensional arrays

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

También puede mezclar la indexación con números enteros individuales con la indexación usando “slicing”. Sin embargo, hacerlo producirá una matriz de rango más bajo que la matriz original.

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #          [ 6]
                             #          [10]] (3, 1)"
```

Array indización con numeros enteros individuales: cuando se indexan en matrices numpy utilizando “slicing”, la matriz resultante siempre será un sub-arreglo de la matriz original. En contraste, la indexación de matrices enteras le permite construir matrices arbitrarias utilizando los datos de otra matriz. Ejemplo:

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

Un truco útil con la indexación de matrices con enteros es seleccionar o mutar un elemento de cada fila de una matriz:

```
import numpy as np

# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a)  # prints "array([[ 1,  2,  3],
          #                [ 4,  5,  6],
          #                [ 7,  8,  9],
          #                [10, 11, 12]])"

# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"

# Mutate one element from each row using the indices in b
a[np.arange(4), b] += 10

print(a)  # prints "array([[11,  2,  3],
          #                [ 4,  5, 16],
          #                [17,  8,  9],
          #                [10, 21, 12]])
```

Indexación de arreglos con booleanos: permite seleccionar elementos **arbitrarios** de una matriz. Con frecuencia, este tipo de indexación se utiliza para seleccionar los elementos de una matriz que satisfacen alguna condición.

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

Ejemplo con bools: “masking”

```
x = np.array([1,6,4,7,9,3,1,5,6,7,3,4,4,3])
x[[1,2,4,3,3,2]]

array([6, 4, 9, 7, 7, 4])
```

which is returning a new array composed of elements 1, 2, 4, etc from the original array.

Alternatively, one can also pass a boolean array of `True/False` values, called a **mask**, indicating which items to keep:

```
x[np.array([True, False, False, True, True, True, False, False, True, True,
True, False, False, True])]

array([1, 7, 9, 3, 6, 7, 3, 3])
```

```python
import numpy as np
import matplotlib.pyplot as plt

plt.figure(1)
plt.clf()
plt.axis([-10, 10, -10, 10])

# Define properties of the "bouncing balls"
n = 10
pos = (20 * np.random.sample(n*2) - 10).reshape(n, 2)
vel = (0.3 * np.random.normal(size=n*2)).reshape(n, 2)
sizes = 100 * np.random.sample(n) + 100

# Colors where each row is (Red, Green, Blue, Alpha).  Each can go
# from 0 to 1.  Alpha is the transparency.
colors = np.random.sample([n, 4])

# Draw all the circles and return an object ``circles`` that allows
# manipulation of the plotted circles.
circles = plt.scatter(pos[:,0], pos[:,1], marker='o', s=sizes, c=colors)

for i in range(100):
    pos = pos + vel
    bounce = abs(pos) > 10       # Find balls that are outside walls
    vel[bounce] = -vel[bounce]   # Bounce if outside the walls
    circles.set_offsets(pos)     # Change the positions
    plt.draw()
    plt.show()
```

“masking”: ejemplos con condiones

```
x[x > 3.4]
```

```
array([6, 4, 7, 9, 5, 6, 7, 4, 4])
```

```
Condiciones se pueden combinar:
x[(x > 3.4) & (x < 5.5)]
```

```
array([4, 5, 4, 4])
```

```
Por supuesto, la máscara booleana se puede derivar de un
arreglo diferente a x siempre que sea del tamaño
correcto:
```

```
x = np.linspace(-1., 1., 14)
y = np.array([1,6,4,7,9,3,1,5,6,7,3,4,4,3])

y[(x > -0.5) & (x < 0.4)]
```

```
array([9, 3, 1, 5, 6, 7])
```

Ejemplo con bools: “masking”

Dado que la máscara en sí misma es una matriz, puede almacenarse en una variable y usarse como una máscara para diferentes matrices::

```
keep = (x > -0.5) & (x < 0.4)
x_new = x[keep]
y_new = y[keep]
```

```
x_new
```

```
array([-0.38461538, -0.23076923, -0.07692308,  0.07692308,  0.23076923,
0.38461538])
```

```
y_new
```

```
array([9, 3, 1, 5, 6, 7])
```

También puede aparecer una máscara en el lado izquierdo de una asignación:

```
y[y > 5] = 0.
array([1, 0, 4, 0, 0, 3, 1, 5, 0, 0, 3, 4, 4, 3])
```

Valores NaN

En las matrices/arreglos, algunos de los valores son a veces NaN, lo que significa que no es un número.

Not **a** **N**umber = NaN

Si multiplican un valor de NaN por otro valor, obtiene NaN.

Y si hay algún valor de NaN en una suma, el resultado total será NaN.

Una forma de evitar esto es usar np.nansum en lugar de np.sum para encontrar la suma:

```
x = np.array([1,2,3,np.nan])

np.nansum(x)

6.0

np.nanmax(x)

3.0
```

Valores NaN

También pueden usar `np.isnan` para averiguar dónde están los valores `NaN`.

Por ejemplo, el arreglo

```
[~np.isnan(array)]
```

devolverá todos los valores que no son NaN
(porque “~” significa ‘no')

```
x = np.array([1,2,3,np.nan])

np.isnan(x)

array([False, False, False,  True], dtype=bool)

x[np.isnan(x)]

array([ nan])

x[~np.isnan(x)]

array([ 1.,  2.,  3.])
```

Funciones

Además de una clase de matriz, Numpy contiene una serie de funciones **vectorizadas**, lo que significa que las **funciones pueden actuar sobre todos los elementos de una matriz**, por lo general mucho más rápido de lo que podría lograrse haciendo un “loop” sobre la matriz o arreglo.

Funciones

```
theta = np.linspace(0., 2. * np.pi, 10)

theta

array([ 0.        ,  0.6981317 ,  1.3962634 ,  2.0943951 ,  2.7925268 ,
        3.4906585 ,  4.1887902 ,  4.88692191,  5.58505361,  6.28318531])

np.sin(theta)

array([  0.00000000e+00,   6.42787610e-01,   9.84807753e-01,
         8.66025404e-01,   3.42020143e-01,  -3.42020143e-01,
        -8.66025404e-01,  -9.84807753e-01,  -6.42787610e-01,
        -2.44929360e-16])
```

```
# uniform distribution between 0 and 1
np.random.random(10)

array([ 0.01643346,  0.32585845,  0.27758201,  0.55505831,  0.55628474,
        0.58441864,  0.23725591,  0.39776078,  0.73614927,  0.91676155])
```

```
# 10 values from a gaussian distribution with mean 3 and sigma 1
np.random.normal(3., 1., 10)

array([ 3.66836993,  2.93423217,  1.88696003,  2.15840383,  4.18619608,
        3.06480279,  4.11898749,  0.6682424 ,  3.88342969,  1.59797832])
```

La función `sum` es muy util.

```
import numpy as np

x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

Mas funciones y operaciones:

https://docs.scipy.org/doc/numpy/reference/routines.math.html

Trigonometric functions

sin(x, /[, out, where, casting, order,
cos(x, /[, out, where, casting, order,
tan(x, /[, out, where, casting, order,
arcsin(x, /[, out, where, casting, ord
arccos(x, /[, out, where, casting, orc
arctan(x, /[, out, where, casting, orc
hypot(x1, x2, /[, out, where, casting
arctan2(x1, x2, /[, out, where, casti
degrees(x, /[, out, where, casting, o
radians(x, /[, out, where, casting, or
unwrap(p[, discont, axis])
deg2rad(x, /[, out, where, casting, c
rad2deg(x, /[, out, where, casting, c

Hyperbolic functions

sinh(x, /[, out, where, casting, order
cosh(x, /[, out, where, casting, orde
tanh(x, /[, out, where, casting, orde
arcsinh(x, /[, out, where, casting, or
arccosh(x, /[, out, where, casting, o
arctanh(x, /[, out, where, casting, o

Rounding

around(a[, decimals, out])
round_(a[, decimals, out])
rint(x, /[, out, where, casting, order,
fix(x[, out])
floor(x, /[, out, where, casting, orde
ceil(x, /[, out, where, casting, order,
trunc(x, /[, out, where, casting, orde

Other special functions

i0(x)	Modified Bessel function of the first kind, order
sinc(x)	Return the sinc function.

Floating point routines

signbit(x, /[, out, where, casting, order, ...])	Returns el
copysign(x1, x2, /[, out, where, casting, ...])	Change th
frexp(x[, out1, out2], / [[, out, where, ...])	Decompos
ldexp(x1, x2, /[, out, where, casting, ...])	Returns x1
nextafter(x1, x2, /[, out, where, casting, ...])	Return the
spacing(x, /[, out, where, casting, order, ...])	Return the

Rational routines

lcm(x1, x2, /[, out, where, casting, order, ...])	Returns th
gcd(x1, x2, /[, out, where, casting, order, ...])	Returns th

Arithmetic operations

add(x1, x2, /[, out, where, casting, order, ...])	Add argu
reciprocal(x, /[, out, where, casting, ...])	Return th
positive(x, /[, out, where, casting, order, ...])	Numerica
negative(x, /[, out, where, casting, order, ...])	Numerica
multiply(x1, x2, /[, out, where, casting, ...])	Multiply a
divide(x1, x2, /[, out, where, casting, ...])	Returns a
power(x1, x2, /[, out, where, casting, ...])	First arra
	wise.
subtract(x1, x2, /[, out, where, casting, ...])	Subtract
true_divide(x1, x2, /[, out, where, ...])	Returns a
floor_divide(x1, x2, /[, out, where, ...])	Return th
	inputs.
float_power(x1, x2, /[, out, where, ...])	First arra

Sums, products, differences

prod(a[, axis, dtype, out, keepdims, initial])	Retu
sum(a[, axis, dtype, out, keepdims, initial])	Sum
nanprod(a[, axis, dtype, out, keepdims])	Retu
	Num
nansum(a[, axis, dtype, out, keepdims])	Retu
cumprod(a[, axis, dtype, out])	Num
cumsum(a[, axis, dtype, out])	Retu
nancumprod(a[, axis, dtype, out])	Retu
	Retu
nancumsum(a[, axis, dtype, out])	treati
	Retu
diff(a[, n, axis])	Not a
ediff1d(ary[, to_end, to_begin])	Calcu
gradient(f, *varargs, **kwargs)	The c
cross(a, b[, axisa, axisb, axisc, axis])	Retu
trapz(y[, x, dx, axis])	Retu
	Integ

Exponents and logarithms

exp(x, /[, out, where, casting, order, ...])	Ca
expm1(x, /[, out, where, casting, order, ...])	Ca
exp2(x, /[, out, where, casting, order, ...])	Ca
log(x, /[, out, where, casting, order, ...])	Na
log10(x, /[, out, where, casting, order, ...])	Re
log2(x, /[, out, where, casting, order, ...])	Ba
log1p(x, /[, out, where, casting, order, ...])	Re
	wi
logaddexp(x1, x2, /[, out, where, casting, ...])	Lo
logaddexp2(x1, x2, /[, out, where, casting, ...])	Lo

fmod(x1, x2, /[, out, where, casting, ...])	Return the element-wise remainder of division.
mod(x1, x2, /[, out, where, casting, order, ...])	Return element-wise remainder of division.
modf(x[, out1, out2], / [[, out, where, ...])	Return the fractional and integral parts of an array
remainder(x1, x2, /[, out, where, casting, ...])	Return element-wise remainder of division.
divmod(x1, x2[, out1, out2], / [[, out, ...])	Return element-wise quotient and remainder simu

Handling complex numbers

angle(z[, deg])	Return the angle of the complex argument.
real(val)	Return the real part of the complex argument.
imag(val)	Return the imaginary part of the complex argument.
conj(x, /[, out, where, casting, order, ...])	Return the complex conjugate, element-wise.

Miscellaneous

convolve(a, v[, mode])	Returns the discrete, linear convolution of two one sequences.
clip(a, a_min, a_max[, out])	Clip (limit) the values in an array.
sqrt(x, /[, out, where, casting, order, ...])	Return the non-negative square-root of an array, e
cbrt(x, /[, out, where, casting, order, ...])	Return the cube-root of an array, element-wise.
square(x, /[, out, where, casting, order, ...])	Return the element-wise square of the input.
absolute(x, /[, out, where, casting, order, ...])	Calculate the absolute value element-wise.
fabs(x, /[, out, where, casting, order, ...])	Compute the absolute values element-wise.
sign(x, /[, out, where, casting, order, ...])	Returns an element-wise indication of the sign of a
heaviside(x1, x2, /[, out, where, casting, ...])	Compute the Heaviside step function.
maximum(x1, x2, /[, out, where, casting, ...])	Element-wise maximum of array elements.
minimum(x1, x2, /[, out, where, casting, ...])	Element-wise minimum of array elements.
fmax(x1, x2, /[, out, where, casting, ...])	Element-wise maximum of array elements.
fmin(x1, x2, /[, out, where, casting, ...])	Element-wise minimum of array elements.
nan_to_num(x[, copy])	Replace NaN with zero and infinity with large finite
real_if_close(a[, tol])	If complex input returns a real array if complex par
interp(x, xp, fp[, left, right, period])	One-dimensional linear interpolation.

“Broadcasting”
and
universal functions

https://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs

“Broadcasting”
and
universal functions

“Broadcasting” es un mecanismo poderoso que permite que Númpy funcione con matrices de diferentes formas al realizar operaciones aritméticas.

Con frecuencia tenemos una matriz más pequeña y una matriz más grande, y queremos usar la matriz más pequeña varias veces para realizar alguna operación en la matriz más grande.

Por ejemplo, supongamos que queremos agregar un vector constante a cada fila de una matriz.

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

Esto funciona; sin embargo, cuando la matriz **x** es muy grande, el cálculo de un loop explícito en Python podría ser muy lento.

Tengan en cuenta que agregar el vector **v** a cada fila de la matriz **x** equivale a formar una matriz **vv** al amontonar varias copias de **v** verticalmente, luego realizar una suma elemental de **x** y **vv**. Podríamos implementar este enfoque de esta manera:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]]"
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #         [ 5  5  7]
          #         [ 8  8 10]
          #         [11 11 13]]"
```

Numpy “broadcasting" nos permite realizar este cálculo sin crear realmente múltiples copias de v:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting

print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

“Broadcasting” de dos matrices/arreglos sigue estas reglas:

- Si las matrices no tienen el mismo rango, anteponga la forma de la matriz de rango inferior con 1s hasta que ambas formas tengan la misma longitud.

- Se dice que las dos matrices son compatibles en una dimensión si tienen el mismo tamaño en la dimensión, o si una de las matrices tiene el tamaño 1 en esa dimensión.

- Los arreglos se pueden transmitir juntos si son compatibles en todas las dimensiones.

- Después de la difusión, cada matriz se comporta como si tuviera una forma igual al máximo elemental de las dos matrices de entrada.

- En cualquier dimensión donde una matriz tenía un tamaño 1 y la otra matriz tenía un tamaño mayor que 1, la primera matriz se comporta como si se hubiera copiado a lo largo de esa dimensión

https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

wiki.scipy.org/EricsBroadcastingDoc

Las funciones que admiten “broadcasting” se conocen como funciones universales. Puede encontrar la lista de todas las funciones universales en:

https://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs

Broadcasting typically makes your code more concise and faster, so you should strive to use it where possible.

“Broadcasting” típicamente hace que tu código sea mas conciso y rápido. Por lo tanto, deberían usarlo cuando sea posible.

Usando Numpy para leer y escribir archivos.

IO in Numpy

https://astrofrog.github.io/py4sci/_static/10.%20Introduction%20to%20Numpy.html

numpy.loadtxt es extremadamente util para datos en formato de columnas.

Ejemplo de datos:

```
1995.00274 0.944444
1995.00548 -1.61111
1995.00821 -3.55556
1995.01095 -9.83333
1995.01369 -10.2222
1995.01643 -9.5
1995.01916 -10.2222
1995.02190 -6.61111
1995.02464 -2.94444
1995.02738 1.55556
1995.03012 0.277778
1995.03285 -1.44444
1995.03559 -3.61111
```

https://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html

(this link contains examples)

```
data = np.loadtxt('data/columns.txt')
data
```

```
array([[  1.99500274e+03,   9.44444000e-01],
       [  1.99500548e+03,  -1.61111000e+00],
       [  1.99500821e+03,  -3.55556000e+00],
       [  1.99501095e+03,  -9.83333000e+00],
       [  1.99501369e+03,  -1.02222000e+01],
       [  1.99501643e+03,  -9.50000000e+00],
       [  1.99501916e+03,  -1.02222000e+01],
       [  1.99502190e+03,  -6.61111000e+00],
       [  1.99502464e+03,  -2.94444000e+00],
       [  1.99502738e+03,   1.55556000e+00],
       [  1.99503012e+03,   2.77778000e-01],
       [  1.99503285e+03,  -1.44444000e+00],
       [  1.99503559e+03,  -3.61111000e+00]])
```

para leer columnas individuales

```
date, temperature = np.loadtxt('data/columns.txt', unpack=True)

date

array([ 1995.00274,  1995.00548,  1995.00821,  1995.01095,  1995.01369,
        1995.01643,  1995.01916,  1995.0219 ,  1995.02464,  1995.02738,
        1995.03012,  1995.03285,  1995.03559])

temperature

array([  0.944444,  -1.61111 ,  -3.55556 ,  -9.83333 , -10.2222  ,
        -9.5     , -10.2222  ,  -6.61111 ,  -2.94444 ,   1.55556 ,
         0.277778,  -1.44444 ,  -3.61111 ])
```

numpy.savetxt : para escribir archivos

`numpy.savetxt`(*fname*, *X*, *fmt='%.18e'*, *delimiter=' '*, *newline='\n'*, *header=''*, *footer=''*, *comments='# '*, *encoding=None*)

https://docs.scipy.org/doc/numpy/reference/generated/numpy.savetxt.html#numpy.savetxt

numpy.savetxt : para escribir archivos

`numpy.savetxt`(*fname*, *X*, *fmt='%.18e'*, *delimiter=' '*, *newline='\n'*, *header=''*, *footer=''*, *comments='# '*, *encoding=None*)

Ejemplos:

```
>>> x = y = z = np.arange(0.0,5.0,1.0)
>>> np.savetxt('test.out', x, delimiter=',')   # X is an array
>>> np.savetxt('test.out', (x,y,z))   # x,y,z equal sized 1D arrays
>>> np.savetxt('test.out', x, fmt='%1.4e')   # use exponential notation
```

https://docs.scipy.org/doc/numpy/reference/generated/numpy.savetxt.html#numpy.savetxt

Mucho mas sobre Numpy:

https://docs.scipy.org/doc/numpy/reference/

Notas mas genéricas sobre IO de archivos .txt:

https://astrofrog.github.io/py4sci/_static/08.%20Reading%20and%20writing%20files.html

Notas mas genéricas sobre IO de archivos .txt: leyendo datos

```
f = open('data/data.txt', 'r')
```

La función open esta tomando el archivo data / data.txt, para abrirlo y devolver un objeto (que llamamos “f”) que luego se puede usar para acceder a los datos.

Tenga en cuenta que f no es la información en el archivo, es lo que se llama un **identificador de archivo (file pointer)**, que apunta al archivo:

```
type(f)

_io.TextIOWrapper
```

Notas mas genéricas sobre IO de archivos .txt: leyendo datos

Para acceder a los datos, simplemente usen read():

```
f.read()
```

La función read () básicamente solo lee el archivo completo y coloca el contenido dentro de una cadena de caracteres. El cual es un formato sin utilidad para computaciones científicas. Necesitamos números.

```
'RAJ        DEJ                          Jmag   e_Jmag\n2000 (deg) 2000 (deg) 2MASS
(mag)  (mag) \n---------- ---------- ----------------- ------ ------\n010.684737 +41.269035
00424433+4116085   9.453  0.052\n010.683469 +41.268585 00424403+4116069   9.321  0.022\n010.685657
+41.269550 00424455+4116103  10.773  0.069\n010.686026 +41.269226 00424464+4116092   9.299
0.063\n010.683465 +41.269676 00424403+4116108  11.507  0.056\n010.686015 +41.269630 00424464+4116106
9.399  0.045\n010.685270 +41.267124 00424446+4116016  12.070  0.035\n'
```

Intentemos esto de nuevo:

```
f.read()
''
```

¿Qué ha pasado? Parece que ya no hay datos?! Lo que ha pasados es que leemos el archivo, y el archivo 'pointer' ahora se encuentra al final del archivo, y no queda nada para leer.

Tenemos que intentar de nuevo para capturar la información dentro de archive de interés.

```
f = open('data/data.txt', 'r')
data = f.read()
```

Ahora data es un “string” continuo de caracteres… Mejor, pero aun no es util porque lo que necesitamos es cada linea y columna de números para hacer operaciones de matrices y arreglos con ellos.

```
'RAJ        DEJ                          Jmag   e_Jmag\n2000 (deg) 2000 (deg) 2MASS
(mag)  (mag) \n---------- ---------- ----------------- ------ ------\n010.684737 +41.269035
00424433+4116085   9.453  0.052\n010.683469 +41.268585 00424403+4116069   9.321  0.022\n010.685657
+41.269550 00424455+4116103  10.773  0.069\n010.686026 +41.269226 00424464+4116092   9.299
0.063\n010.683465 +41.269676 00424403+4116108  11.507  0.056\n010.686015 +41.269630 00424464+4116106
9.399  0.045\n010.685270 +41.267124 00424446+4116016  12.070  0.035\n'
```

```
f = open('data/data.txt', 'r')
for line in f:
    print(repr(line))
```

```
'RAJ         DEJ                            Jmag   e_Jmag\n'
'2000 (deg) 2000 (deg) 2MASS                (mag)  (mag) \n'
'---------- ---------- ------------------ ------ ------\n'
'010.684737 +41.269035 00424433+4116085   9.453  0.052\n'
'010.683469 +41.268585 00424403+4116069   9.321  0.022\n'
'010.685657 +41.269550 00424455+4116103  10.773  0.069\n'
'010.686026 +41.269226 00424464+4116092   9.299  0.063\n'
'010.683465 +41.269676 00424403+4116108  11.507  0.056\n'
'010.686015 +41.269630 00424464+4116106   9.399  0.045\n'
'010.685270 +41.267124 00424446+4116016  12.070  0.035\n'
```

`repr()` muestra cualquier carácter invisible (esto será útil en un minuto).

También tengan en cuenta que ahora estamos haciendo un loop sobre un archivo en lugar de una lista, y esto por definición lee automáticamente la siguiente línea en cada iteración.
Cada línea se devuelve como una cadena.
Observe el \n al final de cada línea: este es un carácter de retorno de línea, que indica el final de una línea.

Ahora estamos leyendo un archivo línea por línea, lo que es necesario es obtener valores numéricos de cada linea. Vamos a examinar la última línea en detalle. Si solo escribimos esta, deberíamos ver la última línea que se imprimió en el loop:

```
line

'010.685270 +41.267124 00424446+4116016  12.070  0.035\n'
```

Primero quitamos el “\n” con “strip”

```
 line = line.strip()

 line

 '010.685270 +41.267124 00424446+4116016  12.070  0.035'
```

Ahora podemos usar la función spit para partir el “string” en elementos separados por espacios:

```
columns = line.split()

columns

[‘010.685270', '+41.267124', '00424446+4116016', '12.070', '0.035']
```

Ahora nos interesan algunas columnas en particular (e.g. “name, so the 2MASS column, and the Jmag column):

```
name = columns[2]
jmag = columns[3]

name

'00424446+4116016'

jmag

'12.070'
```

Pero todos los datos están aun en formato de “string”, o sea que aun no son útiles. Tenemos que pasarlos a “floats”.

```
jmag = float(columns[3])

jmag

12.07
```

Y la ultima cosa que necesitamos es que así se puede leer un archivo linea por linea:

```
line = f.readline()
```

```
# Open file
f = open('data/data.txt', 'r')

# Read and ignore header lines
header1 = f.readline()
header2 = f.readline()
header3 = f.readline()

# Loop over lines and extract variables of interest
for line in f:
    line = line.strip()
    columns = line.split()
    name = columns[2]
    jmag = float(columns[3])
    print(name, jmag)
```

```
00424433+4116085 9.453
00424403+4116069 9.321
00424455+4116103 10.773
00424464+4116092 9.299
00424403+4116108 11.507
00424464+4116106 9.399
00424446+4116016 12.07
```