MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Computer Science Department

COMP9086 – Data Processing and Visualization

Week 7: Introduction to Numpy and how to handle errors

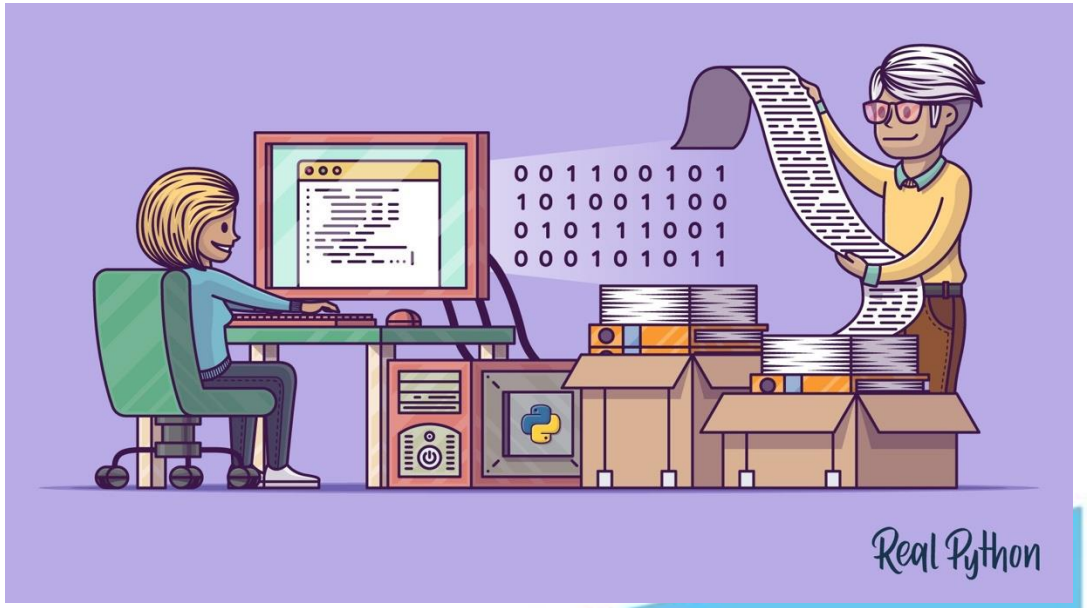
Dr. Bruno Andrade

November, 2025

www.mtu.ie

Last lecture

- The functional paradigm.
- How functions can help us reduce the amount of code needed.



- **Summary:**
  - Introduction to Numpy
  - Operations
  - Basic Array Operations
  - Universal Functions
  - Array Selectors
  - Reading and saving data to files (Numpy Style)
  - Error Handling

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Introduction to NumPy

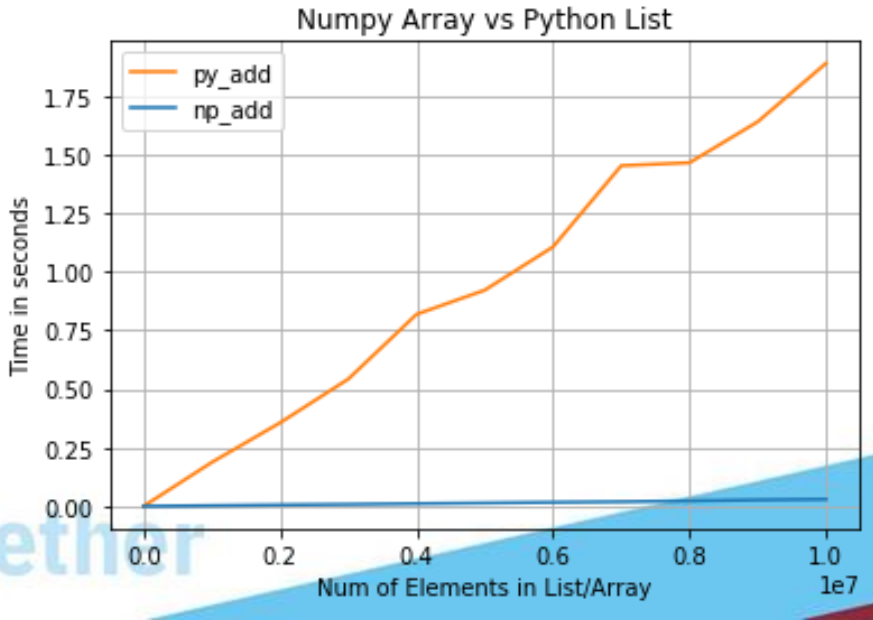
Introduction to Numpy

- **NumPy** is an open-source add-on module to Python that provides routines for **manipulating large arrays** and matrices of numeric data in **pre-compiled**, fast functions.
- At the core of the NumPy package, is the **Ndarray** object.
  - This encapsulates n-dimensional arrays of **homogeneous** data types, with many operations being performed in compiled code for performance.

NumPy arrays

- NumPy arrays facilitate a range of mathematical and other types of operations on large amounts of data.

- NumPy arrays make operations with large amounts of numeric data very fast and are generally much more efficiently than lists.



NumPy arrays

```
import numpy as np
import time
```

```
start = time.time()
a = list(range(10000000))
b = list(range(10000000))
c = []

for i in range(len(a)):
  c.append(a[i] + b[i])
end = time.time()

print(f"{(end-start):.2f} s")
```

```
1.63 s
```

```
#Same process using Numpy
start = time.time()
a = np.arange(10000000, dtype=int)
b = np.arange(10000000, dtype=int)
c = a + b
end = time.time()
print(f"{(end-start):.2f} s")
```

Introduction to Numpy

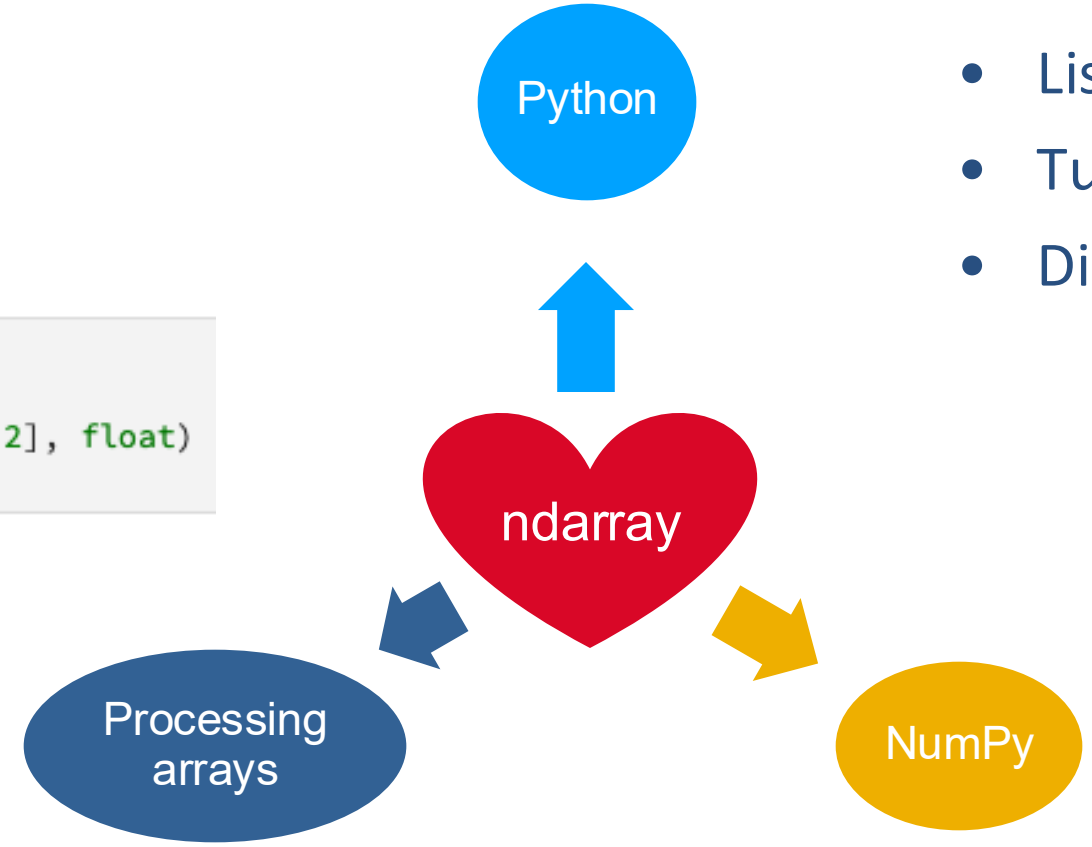
- There are several important differences between NumPy arrays and the standard Python lists
  - NumPy arrays have a **fixed size** at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will **create a new array** and delete the original.
  - The elements in a NumPy array are all required to be of the **same data type**.

Creating Arrays

```
import numpy as np

arr = np.array([5.5, 45.6, 3.2], float)
print (arr)
```

```
[ 5.5 45.6  3.2]
```



- Lists
- Tuples
- Dictionaries

- Replicating
- Joining
- mutating

- Arange
- Ones
- Zeros
- Etc…

NumPy - Arrays

- Array can be accessed using the **square bracket notation** just as with a list

```
import numpy as np

arr = np.array([5.5, 45.6, 3.2], float)
print(f"Individual: {arr[0]}")
arr[0] = 5
print (f"Full Array: {arr}")
```

```
Individual: 5.5
Full Array: [ 5.  45.6  3.2]
```

NumPy – Multi-Dimensional Arrays

- Arrays can be multidimensional. Elements are accessed using **[row, column]** format inside bracket notation

Multidimensional Array:

	0	1	2
0	1	2	3
1	4	5	6

NumPy – Multi-Dimensional Arrays

```
arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print(f"Bidimensional Array: \n {arr}")
print(f"First Row/First Collumn: {arr[0, 0]}")
print(f"Second Row/Third Collumn: {arr[1, 2]}")
```

```
Bidimensional Array:
 [[1. 2. 3.]
 [4. 5. 6.]]
First Row/First Collumn: 1.0
Second Row/Third Collumn: 6.0
```

NumPy – Single Index to 2D Array

- A single index value provided to a multi-dimensional array will refer to an entire row.

```
arr = np.array([[1, 2, 3], [4, 5, 6]], float)
#Changing the first element of the second row
arr[0, 1] = 12.2
print(f"New Bidimensional Array: \n {arr}")
print(f"\nSecond row of the Array: {arr[1]}")
#Changing the values of ALL elements of the first row
arr[0] = 12.2
print(f"\nNew Bidimensional Array: \n {arr}")
```

```
New Bidimensional Array:
 [[ 1.   12.2   3. ]
 [ 4.    5.    6. ]]

Second row of the Array: [4. 5. 6.]

New Bidimensional Array:
 [[12.2 12.2 12.2]
 [ 4.    5.    6. ]]
```

NumPy – Multi-Dimensional Arrays

- We can select a number of columns from a matrix.
- When we select a single column, NumPy flattens the array

```
arr = np.array([[1, 2, 3, 4],
                [4, 5, 6, 7],
                [7, 8, 9, 10],
                [10, 11, 12, 13]], float)
print("Original Array:")
print (arr)

arr2 = arr[:, 1:3]
print("Sliced Array:")
print (arr2)

arr2 = arr[:,3]
print("Last element of each row:")
print (arr2)
```

```
Original Array:
[[ 1.  2.  3.  4.]
 [ 4.  5.  6.  7.]
 [ 7.  8.  9. 10.]
 [10. 11. 12. 13.]]
Sliced Array:
[[ 2.  3.]
 [ 5.  6.]
 [ 8.  9.]
 [11. 12.]]
Last element of each row:
[ 4.  7. 10. 13.]
```

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

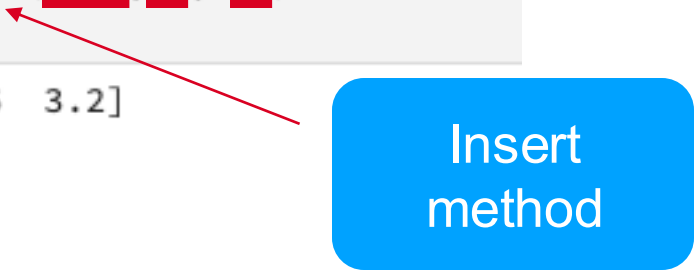
Operations

NumPy - Arrays

- We can *insert* a value at a specific location
  - ***insert****(array, Index, value)*

```
arr1 = np.array([5.5, 45.6, 3.2], float)
arr2 = np.insert(arr1, 1, 99)
print (arr2)
```

```
[ 5.5 99.  45.6  3.2]
```



NumPy - Arrays

- We can *delete* a value at a specific location
  - ***delete****(array, insertIndex)*

```
arr1 = np.array([5.5, 45.6, 3.2], float)
arr2 = np.delete(arr1, 1)
print (arr2)
```

```
[5.5 3.2]
```

Delete method

NumPy - Arrays

- We can use numpy.concatenate to add **one NumPy array to another** Numpy Array

```
import numpy as np

arr1 = np.array([5.5, 45.6, 3.2], float)
arr2 = np.array([3.5, 4.1, 8.4], float)
arr3 = np.concatenate((arr1, arr2))
print (arr3)
```

```
[ 5.5 45.6  3.2  3.5  4.1  8.4]
```

- Note: The concatenate will return a new independent copy of the data.

NumPy – Slicing Operations

- We can use slicing effectively to extract a subset of items from a NumPy array. We can slice along each axis.
- Use of a single ":" in a dimension indicates the use of everything along that dimension

```
arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print(f"Full Array: \n {arr}")

arr2 = arr[0,:]
print(f"\nSlicing rowns: {arr2}")

arr3 = arr[:,0]
print(f"\nSlicing collumns: {arr3}")
```

```
Full Array:
 [[1. 2. 3.]
 [4. 5. 6.]]

Slicing rowns: [1. 2. 3.]

Slicing collumns: [1. 4.]
```

Slicing Arrays

- The following code will retrieve the elements in row 0 and 1 and the elements in column 0 and 1

```
arr = np.array([[14.4, 2.4, 3.5],
                [54.3, 34.4, 98.22],
                [100, 200, 300]], float)
print("Original Array:")
print(arr)
print("Sliced Array:")
print (arr[0:2, 0:2])
```

```
Original Array:
[[ 14.4    2.4    3.5 ]
 [ 54.3   34.4   98.22]
 [100.   200.   300.  ]]
Sliced Array:
[[14.4  2.4]
 [54.3 34.4]]
```

Important consideration when slicing!!

- When we use slicing on **lists**, it returns a new list.
- However, when performing slicing on a NumPy array it will return **a view of the original array**.

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

resultA = data[:,0]
resultA[0] = 200
print (data)
```

```
[[200.   2.   3.]
 [  2.   4.   5.]
 [  4.   5.   7.]
 [  6.   2.   3.]]
```

Copying a NumPy Object

- To copy a NumPy array use **NumPy.copy** function, which takes in as an argument the array you want to copy

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

dataCopyA = data
dataCopyA[0,0] = 200

print (data)
```

```
[[200.   2.   3.]
 [  2.   4.   5.]
 [  4.   5.   7.]
 [  6.   2.   3.]]
```

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

dataCopyA = np.copy(data)
dataCopyA[0,0] = 200

print (data)
```

```
[[1. 2. 3.]
 [2. 4. 5.]
 [4. 5. 7.]
 [6. 2. 3.]]
```

Use of len Function in Arrays

- len function can be used to obtain the number of rows or the number of columns
  - len of 2D array will return the number of rows
  - len of 2D row will return the number of columns within that row

```
arr = np.array([[14.4, 2.4, 56.4],
                [54.3, 34.4, 98.22]], float)

print (f"Array size: {len(arr)}")
print (f"First row size: {len(arr[0])}")
```

```
Array size: 2
First row size: 3
```

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Numpy in-built functions for array creation

Other Ways to Create Arrays

- The *arange* function is similar to the range function but returns an numpy array. However, unlike the range function, the arrange function immediately create the NumPy array.

```
arr1 = np.arange(5, dtype=float)
arr2 = np.arange(1, 6, 2, dtype=int)

print (arr1)
print (arr2)
```

```
[0. 1. 2. 3. 4.]
[1 3 5]
```

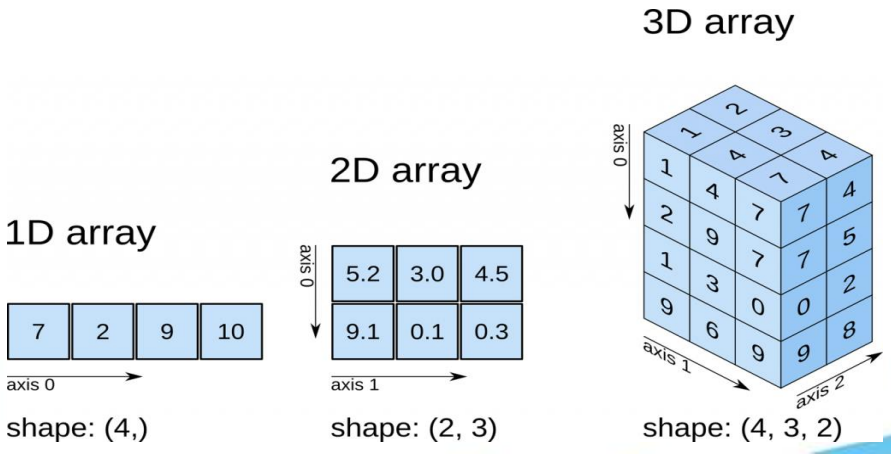
- It's only possible to create 1D array with arrange
  - But we can reshape arrays (see next slide)

Reshaping Arrays

- Arrays can be reshaped by specifying new dimensions.
  - In the example, we turn a ten-element one-dimensional array into a two-dimensional 10x10 elements:
  - The size of the new array must match exactly the size of the original array.

```
arr1 = np.arange(0,100, dtype=float)
arr2 = arr1.reshape((10, 10))
print (arr2)
```

```
[[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14. 15. 16. 17. 18. 19.]
 [20. 21. 22. 23. 24. 25. 26. 27. 28. 29.]
 [30. 31. 32. 33. 34. 35. 36. 37. 38. 39.]
 [40. 41. 42. 43. 44. 45. 46. 47. 48. 49.]
 [50. 51. 52. 53. 54. 55. 56. 57. 58. 59.]
 [60. 61. 62. 63. 64. 65. 66. 67. 68. 69.]
 [70. 71. 72. 73. 74. 75. 76. 77. 78. 79.]
 [80. 81. 82. 83. 84. 85. 86. 87. 88. 89.]
 [90. 91. 92. 93. 94. 95. 96. 97. 98. 99.]]
```



Other Ways to Create Arrays

- The functions zeros and ones create new arrays of specified dimensions filled with these values
  - Commonly used functions to create new arrays

```
arr1 = np.ones((2,3), dtype=float)
arr2 = np.zeros(7, dtype=int)
print (arr1)
print (arr2)
```

```
[[ 1.  1.  1.]
 [ 1.  1.  1.]]

[0 0 0 0 0 0 0]
```

NumPy – Appending to Arrays

- We can add elements using append to arrays in NumPy
  - numpy.**append**(*arr, values, axis=None*)

```
arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print (arr)

arr1 = np.append(arr, [[7, 8, 9]])

print (arr1)
```

```
[[1. 2. 3.]
 [4. 5. 6.]]
[1. 2. 3. 4. 5. 6. 7. 8. 9.]
```

- arr - Values are appended to a copy of this array.
- values - These values are appended to a copy of arr
- axis = The axis along which values are appended.

If axis is not defined, both arr and values are flattened before use.

NumPy – Multi-Dimensional Arrays

```
arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print (arr)

arr1 = np.append(arr, [[7, 8, 9]], axis = ?)

print (arr1)
```

axis = 0 refers to the vertical axis

axis = 1 refers to the horizontal axis

Dimension of values being added must be same as the specific axis we are adding to

NumPy – Multi-Dimensional Arrays

```
arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print (arr)

arr1 = np.append(arr, [[7, 8, 9]], axis = 0)

print (arr1)
```

Add a row containing the values [7, 8, 9] to axis = 0

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

[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]]
```

NumPy – Multi-Dimensional Arrays

```
arr = np.array([[1, 2, 3],
                [4, 5, 6]], float)
print (arr)

arr1 = np.append(arr, [[7, 8, 9]], axis = 1)

print (arr1)
```

```
[[1. 2. 3.]
 [4. 5. 6.]]
---------------------------------------------------------
ValueError                                Traceback
/var/folders/9x/hx0451rj4n1fcxsm1m6ch5hx7p3m0h/T/ip                in <module>
```

Add a column containing the values [7, 8, 9] to axis = 1

Generates an error specifying array dimensions don't match because each column only contains two values (not three)

[[1. 2. 3.]
[4. 5. 6.]]

NumPy – Multi-Dimensional Arrays

```
arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print (arr)

arr1 = np.append(arr, [[7], [8]], axis = 1)

print (arr1)
```

We used two [] brackets, that is because we are adding a single column element to each row

[[1. 2. 3.]
 [4. 5. 6.]]

[[1. 2. 3. 7.]
 [4. 5. 6. 8.]]

Obtaining Max or Min in an Array

- numpy.amin(array, axis)
  - Return the minimum of an array or minimum along an axis.
- numpy.amax(array, axis)
  - Return the maximum of an array or maximum along an axis.

```
arr1 = np.array([10,20,30], float)
arr2 = np.array([1,2,3], float)

print (np.amax(arr1))
print (np.amax(arr2))
```

```
30.0
3.0
```

Obtain Max or Min in an Array

- For multi-dimensional arrays we can specify the axis.
  - If we don't specify the axis it will determine the maximum for the entire array

```
arr1 = np.array([[10,20,30],[50, 60, 10]], float)
print (arr1)

print (np.amax(arr1))

print (np.amax(arr1, axis=0))

print (np.amax(arr1, axis=1))
```

```
[[10. 20. 30.]
 [50. 60. 10.]]
60.0
[50. 60. 30.]
[30. 60.]
```

Biggest element

Biggest elements in a 3 element row

Biggest elements in each column

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Basic Array Operations

Basic Array Operations

- Many functions exist for extracting whole-array properties.
- The items in an array can be summed or multiplied:
- These functions can be performed on multi-dimensional arrays
  - We can also provide an additional element of the axis we wish to access

```
arr1 = np.array([[1, 2, 4],[3, 4, 2]], float)
print (arr1)
print (np.sum(arr1))
print (np.product(arr1))
print (np.sum(arr1, axis = 0))
print (np.mean(arr1, axis = 1))
```

```
[[1. 2. 4.]
 [3. 4. 2.]]
16.0
192.0
[4. 6. 6.]
[2.33333333 3.        ]
```

Basic Array Operations

- A number of routines enable computation of statistical quantities in array datasets, such as the mean (average), variance, and standard deviation.
- You can specify an axis on these operations as well
- Large number of mathematical functional available.

```
arr = np.array([2, 1, 9], float)
print (np.mean(arr))
print (np.var(arr))
print (np.std(arr))
```

```
4.0
12.666666666666666
3.559026084010437
```

You can also specify the axis that we wish to perform these operations on

Array Mathematical Operations

- When standard mathematical operations are used with two arrays, they are applied on an **element by-element** basis.
  - This means that the arrays should be the same size during addition, subtraction,
  - NumPy arrays support the typical range of operators +,-, *, /, %, **

Array Mathematical Operations

- Example

```
arr1 = np.array([10,20,30], float)
arr2 = np.array([1,2,3], float)

print (arr1*arr2)
print (arr1/arr2)
print (arr1**arr2)
```

```
[10. 40. 90.]
[10. 10. 10.]
[1.0e+01 4.0e+02 2.7e+04]
```

Array Mathematical Operations

- For two-dimensional arrays, multiplication remains element-wise and does not correspond to matrix multiplication.

```
arr1 = np.array([[10,20], [30, 40]], float)
arr2 = np.array([[1,2], [3,4]], float)

print (arr1+arr2)
```

```
[[11. 22.]
 [33. 44.]]
```

Array Mathematical Operations

- For matrix multiplication we use the numpy.dot function.

```
arr1 = np.array([10,20,30], float)
arr2 = np.array([1,2,3], float)

print(np.dot(arr1,arr2))
```

```
140.0
```

Linear algebra with Numpy

- Numpy.linalg has a standard set o methods for matrix decompositions, such as inverse and determinant:

```
from numpy.linalg import inv,qr
```

```
X = np.random.standard_normal((5,5))
mat = X.T.dot(X)
```

```
mat
```

```
array([[ 5.41306797, -0.2940687 , -1.98112548,  0.02204812, -1.58538904],
       [-0.2940687 ,  8.15948879, -0.17206254,  2.89824239, -3.77880973],
       [-1.98112548, -0.17206254,  5.26072754,  4.37722225, -0.31364562],
       [ 0.02204812,  2.89824239,  4.37722225,  5.77790349, -2.14796938],
       [-1.58538904, -3.77880973, -0.31364562, -2.14796938,  2.99007472]])
```

```
inv(mat)
```

```
array([[ 2.83984049,  2.87733146,  4.44374903, -3.73254929,  2.92685026],
       [ 2.87733146,  3.26112525,  4.72608392, -4.01611137,  3.25767541],
       [ 4.44374903,  4.72608392,  7.9623086 , -6.83936959,  4.25094051],
       [-3.73254929, -4.01611137, -6.83936959,  6.13133305, -3.36743631],
       [ 2.92685026,  3.25767541,  4.25094051, -3.36743631,  4.03015705]])
```

Linear algebra with Numpy

- Array @ array works like array.dot(array):

```
mat @ inv(mat)
```

```
array([[ 1.00000000e+00,  1.05133010e-15,  7.11871785e-16,
        -3.72337217e-15,  1.36269799e-16],
       [ 3.73254441e-15,  1.00000000e+00,  1.22738441e-15,
        -1.78713241e-15, -1.64732973e-15],
       [-8.49176890e-16, -1.24961593e-15,  1.00000000e+00,
         1.11140235e-16, -1.65108328e-15],
       [ 3.72491064e-15,  1.58677300e-15, -3.89514650e-16,
         1.00000000e+00,  1.72381248e-15],
       [-2.22044605e-15,  0.00000000e+00,  8.88178420e-16,
         4.44089210e-16,  1.00000000e+00]])
```

```
np.rint(mat.dot(inv(mat)))
```

```
array([[ 1.,  0.,  0., -0.,  0.],
       [ 0.,  1.,  0., -0., -0.],
       [-0., -0.,  1.,  0., -0.],
       [ 0.,  0., -0.,  1.,  0.],
       [-0.,  0.,  0.,  0.,  1.]])
```

Array Mathematical Operations

- In addition to the standard operators, NumPy offer a large library of common mathematical functions that can be applied element-wise to arrays.
- abs, sign, sqrt, log2, log10, exp, sin, cos, tan, arcsin, arccos, arctan, floor, ceil, and rint, cumsum, degrees, etc

```
arr1 = np.array([[10,20], [30, 40]], float)
arr2 = np.sqrt(arr1)
arr3 = np.log10(arr1)
arr4 = np.rint(arr2)

print (arr2)
print (arr3)
print (arr4)
```

```
[[3.16227766 4.47213595]
 [5.47722558 6.32455532]]
[[1.         1.30103   ]
 [1.47712125 1.60205999]]
[[3. 4.]
 [5. 6.]]
```

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Universal functions (Ufuncs)

Ufuncs

- Numpy has also a number of functions that performs element-wise operations.
- They are known as Ufuncs. They come in two flavours:

Unary Ufuncs

Binary Ufuncs

Unary Ufuncs

- They are simple element-wise transformations, such as:

```
import numpy as np
arr = np.arange(10)
arr
```

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

```
print(np.sqrt(arr))
print(np.exp(arr))
```

```
[0.         1.         1.41421356 1.73205081 2.         2.23606798
 2.44948974 2.64575131 2.82842712 3.        ]
[1.00000000e+00 2.71828183e+00 7.38905610e+00 2.00855369e+01
 5.45981500e+01 1.48413159e+02 4.03428793e+02 1.09663316e+03
 2.98095799e+03 8.10308393e+03]
```

Unary Ufuncs

Function	Description
abs, fabs	Absolute value element-wise for numbers
sqrt	Compute the square root of each element
square	Square each element (n**2)
exp	Exponent of each element
log, log10, log2, log1p	Natural logarithm, base 10, base 2 and base(1+x)
sign	Sign of each element
ceil	Compute the ceiling of each element
floor	Compute the floor of each element
isnan	Check if the value is NaN
isfinite, isinf	Check if value is finite or not
cos, cos, sin, tan	Do hyperbolic trigonometric stuff
arcsin, arctan, arccos	Do inverse trigonometric stuff

Binary Ufuncs

- Binary Ufuncs take two arrays, as input:

```
x = np.random.standard_normal(8)
#Here we are generating a normal distribution with 8 elements
y = np.random.standard_normal(8)

print(x)
print(y)
```

```
[-1.85643999  0.0151025  -0.52933095 -0.15781995  1.63218273 -0.87338401
 -1.21518343  0.83069138]
[-1.30195242  2.15440929  1.03857988  1.06817169  0.2216669  -0.21530379
 -0.28155318  2.16773777]
```

```
np.maximum(x,y)
```

```
array([-1.30195242,  2.15440929,  1.03857988,  1.06817169,  1.63218273,
       -0.21530379, -0.28155318,  2.16773777])
```

```
np.multiply(x,y)
```

```
array([ 2.41699653,  0.03253697, -0.54975248, -0.16857881,  0.36180089,
        0.18804289,  0.34213876,  1.80072109])
```

Binary Ufuncs

Function	Description
add	Add corresponding elements in array
subtract	Subtract elements in second array from first array
multiply	Multiply array elements
divide	Divide elements
power	Raise elements in first to the power of second array
maximum	Element-wise maximum
minimum	Element-wise minimum
mod	Element-wise modulus (remainder of division)
greater, greater_equal, less, less_equal, equal, not_equal	>, >=, <, <=, ==, !=
logical_and	Element-wise truth value of AND
logical_or	Element-wise truth value of OR
logical_xor	Element-wise truth value of XOR

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Array selectors

Array Selectors

- We have already seen that, like lists, individual elements and slices of arrays can be selected using bracket notation. Unlike lists, however, **arrays also permit selection using other arrays or lists**.

- In NumPy this is referred to as fancy indexing

- That is, we can use an array to filter for specific subsets of elements of other arrays.

Array Selectors – Fancy indexing

```
import numpy as np
rand = np.random.RandomState(42)

x = rand.randint(100, size=10)
print(x)
```

```
[51 92 14 71 60 20 82 86 74 74]
```

```
[x[3], x[7], x[2]]
```

```
[71, 86, 14]
```

```
ind = [3, 7, 4]
x[ind]
```

```
array([71, 86, 60])
```

Array Selectors

- We can use a Boolean array to **filter** the contents of another array.
- Below we use a Boolean array to select a subset of element from the NumPy array

```
arr1 = np.array([45, 3, 2, 5, 67], float)
boolArr1 = np.array([True, False, True, False, True], bool)
print (arr1[boolArr1])
```

```
[45.  2. 67.]
```

[45. 2. 67.]

Notice the program only returns the elements in arr1, where the corresponding element in the Boolean array is true

Array Selectors

```
arr2D = np.array([[45, 3, 67, 34],[12, 43, 73, 36]]
boolArr3 = np.array([True, False], bool)
print (arr2D[boolArr3])


arr2D = np.array([[45, 3, 67, 34],[12, 43, 73, 36]], float)
boolArr3 = np.array([[True, False, True, False],
                     [True, True, False, True]], bool)
print (arr2D[boolArr3])
```

```
[[45.  3. 67. 34.]]
[45. 67. 12. 43. 36.]
```

If we provide a 1D Boolean array as an index to a 2D array the boolean values refer to rows

If we provide a matching Boolean array it will select individual values and return a flat array

Selecting Columns from 2D Array

```
arr2D = np.array([[45, 3, 67],[12, 43, 73]], float)

boolArr4 = np.array([True, False, True], bool)
print (arr2D[:,boolArr4])
```

```
[[45. 67.]
 [12. 73.]]
```

Here we use booleans to select particular columns from a 2D array. We specify all rows using : and we select the first and last column for selection

Comparison operators

```
arr1 = np.array([1, 3, 20, 5, 6, 78], float)
arr2 = np.array([1, 2, 3, 67, 56, 32], float)

resultArr = arr1>arr2
print (arr1[resultArr])
```

```
[ 3. 20. 78.]
```

Here we combine comparison operators and boolean selection. This will print out all those values in arr1 that are greater than the corresponding value in arr2 (very useful)

Comparison Operators

- The following code applies a conditional operator the column with index 1 and returns the rows the satisfy this condition.

```
data = np.array([[1, 2, 3], [2, 4, 5], [4, 5, 7], [6, 2, 3]], float)
print (data)

# return all rows in array where the element at index 1 in a row equals 2
newdata = data[data[:,1] == 2.]
print (newdata)
```

```
[[1. 2. 3.]
 [2. 4. 5.]
 [4. 5. 7.]
 [6. 2. 3.]]
[[1. 2. 3.]
 [6. 2. 3.]]
```

Returns all rows in the 2D array such that the value of the column with index 1 in that row contains the value 2

Comparison operators

- Boolean comparisons can be used to compare members element-wise on arrays of equal size.
- These operators (<,>, >=, <=, ==) return a boolean array as a result

```
arr1 = np.array([1, 3, 0], float)
arr2 = np.array([1, 2, 3], float)

resultArr = arr1>arr2
print (resultArr)
print (arr1== arr2)
```

```
[False  True False]
[ True False False]
```

Comparison Operators

- An Array can be compared to a single value.

```
arr = np.array([1, 3, 0, 2, 4, 5], float)

result1 = arr > 2
result2 = arr == 2
print (result1)
print (result2)
print (arr[result1])
print (arr[result2])
```

```
[False  True False False  True  True]
[False False False  True False False]
[3. 4. 5.]
[2.]
```

Comparison Operators

- When we perform array based indexing it produces a **new copy** of the array (this is unlike slicing which produces a view of the original data)

```
arr2D = np.array([[45, 3, 67],[12, 43, 73],[22, 62, 17]], float)
boolArr2 = np.array([True, False, True], bool)

newArr = arr2D[boolArr2]
print (newArr)

newArr[0, 1] = 50000

print (arr2D)
```

```
[[45.  3. 67.]
 [22. 62. 17.]]
[[45.  3. 67.]
 [12. 43. 73.]
 [22. 62. 17.]]
```

Notice the change made to the newArr array is not relfected in the original array

Logical Operators

- You can combine multiple conditions using logical operators.
- Unlike standard Python the logical operators used are **&** and **|**

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

resultA = data[:,0]>3
resultB = data[:,2]>6

print (data[resultA & resultB])
```

```
[[4. 5. 7.]]
```

Here we combine two conditions using & (we could chain as many conditions as we wish)

Back to Array Selectors

- In addition to Boolean selection, it is possible to select using integer arrays.
- In this example the new array *c* is composed by selecting the elements from *a* using the index specified by the elements of *b*.

```
a = np.array([2, 4, 6, 8], float)
b = np.array([0, 0, 1, 3, 2, 1], int)
c = a[b]
print (c)
```

```
[2. 2. 4. 8. 6. 4.]
```

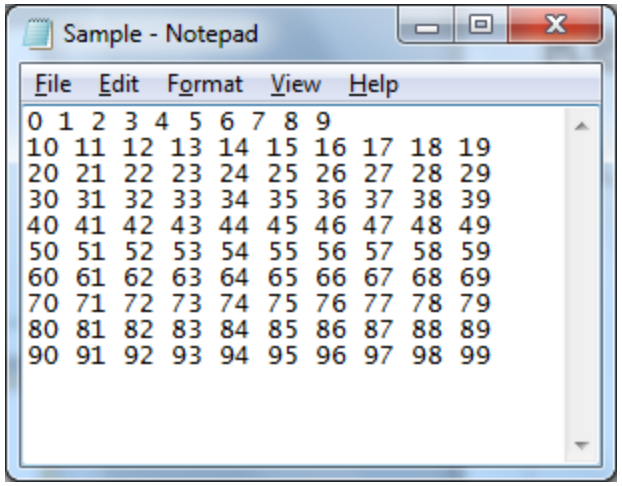
The array c is composed of index 0,0, 1, 3, 2, 1 of the array a

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Reading and saving data to a file (Numpy style)

Reading Data From a File

- How to read that data depicted in the file Sample.txt, into a two-dimension array?
- ***np.genfromtxt*** uses *dtype=float* by default



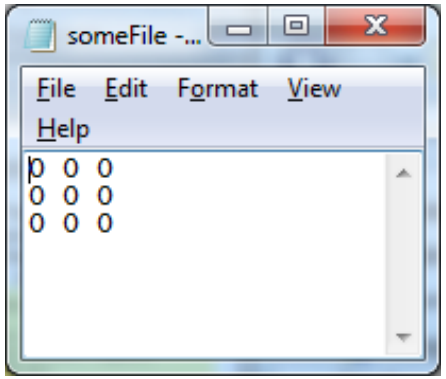

```
data = np.genfromtxt('Datasets/Sample.txt', dtype=int)
print (data[0,0])
print (data[1,0])
print (data[2,0])
print (data[3,0])
```

```
0
10
20
30
```

Saving Data To a File

- Use savetxt method to save data from an array to a file. The code below saves the 2D array data to the file called someFile.txt.

```
data = np.zeros((3,3))
np.savetxt("someFile.txt", data)
```



MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Error Handling

The inevitable error

- You probably saw lots of this by now:

```
---------------------------------------
ZeroDivisionError
Cell In[1], line 1
----> 1 print(10/0)

ZeroDivisionError: division by zero
```

```
---------------------------------------------------------------
TypeError                                 Traceback (most rec
Cell In[2], line 1
----> 1 print("10"/0)

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

- Errors are part of a programmers life, we shouldn't avoid them entirely, but know how to manage them gracefully.

The two types of error (Syntax error vs Exception)

- Syntax error:
  - A grammatical error in a sentence, the code won't even run.
  - e.g. if x=5: (should use == instead of =)
  - e.g. calling an undeclared variable.
  - e.g. calling a function before initialising it.

The two types of error (Syntax error vs Exception)

- Exception:
  - The code is grammatically correct, but a runtime error occurred but you can recover from it.

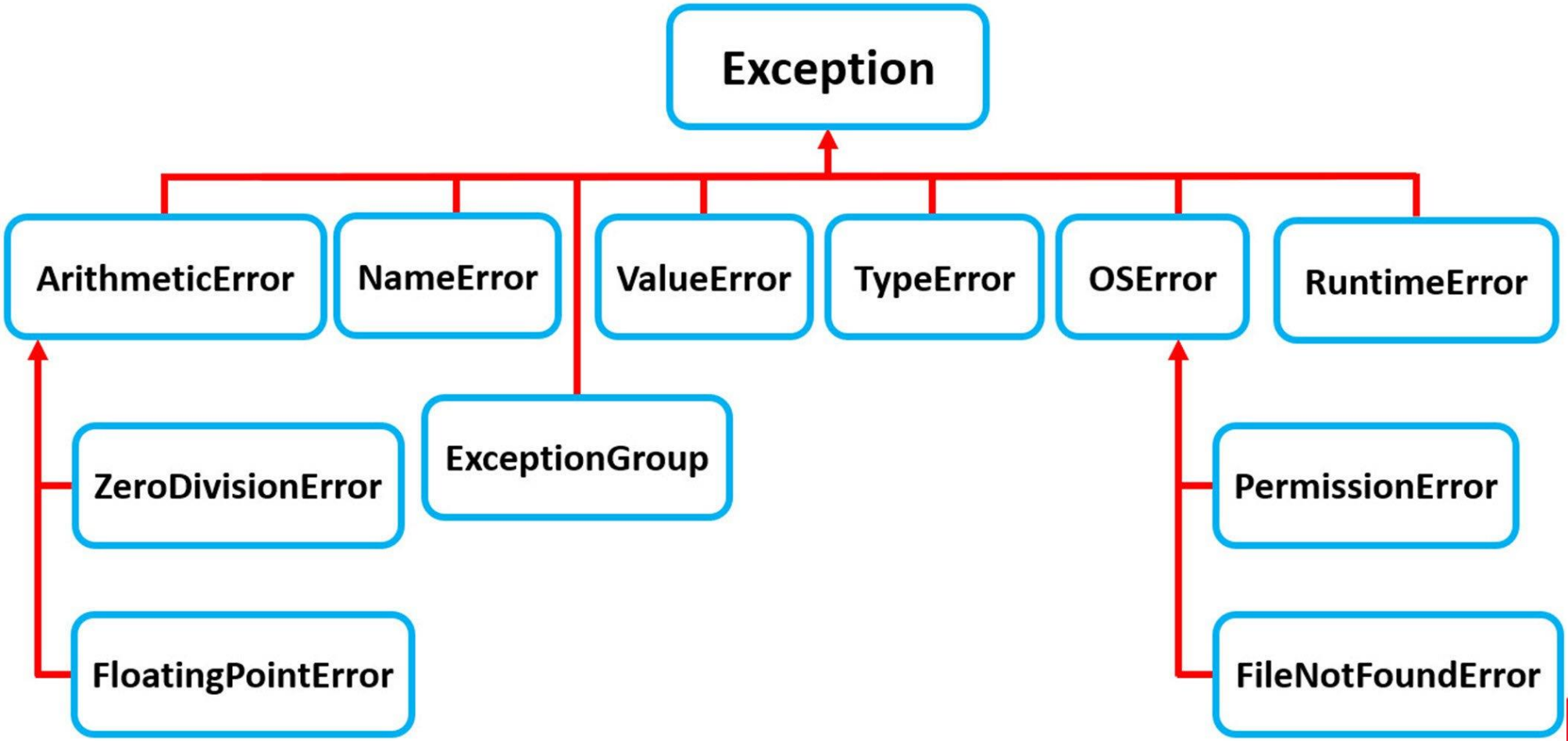
```
---------------------------------
ZeroDivisionError
Cell In[1], line 1
----> 1 print(10/0)

ZeroDivisionError: division by zero
```

```
---------------------------------------------------------------
TypeError                                 Traceback (most rec
Cell In[2], line 1
----> 1 print("10"/0)

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

The two types of error (Syntax error vs Exception)



How to handle it! Using the try…except block

- try…except block:

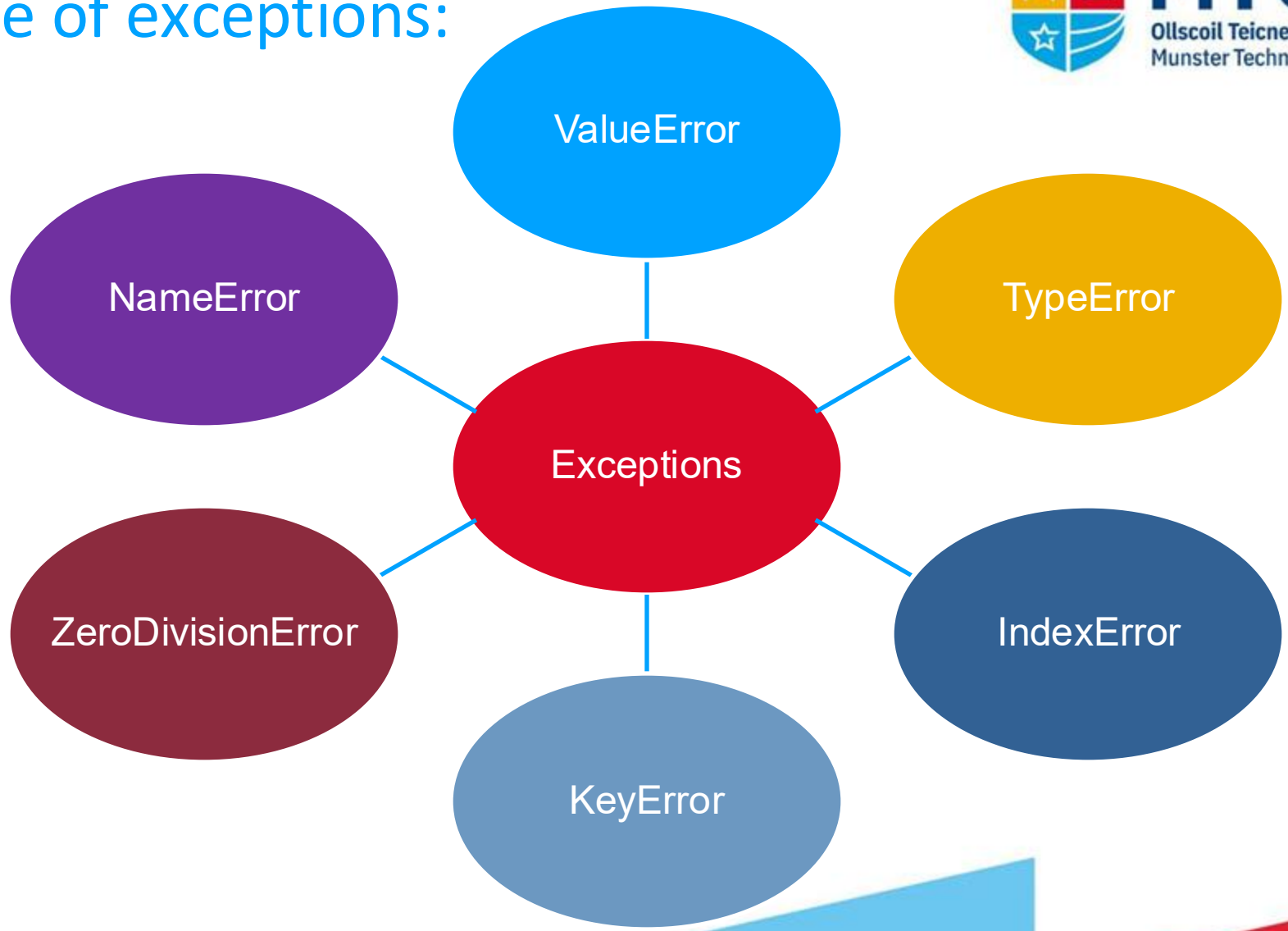
```
try:
    Code to be tested will be placed here
except error:
    Code to let the user know what happened and the options go here.
```

How to handle it! Using the try…except block

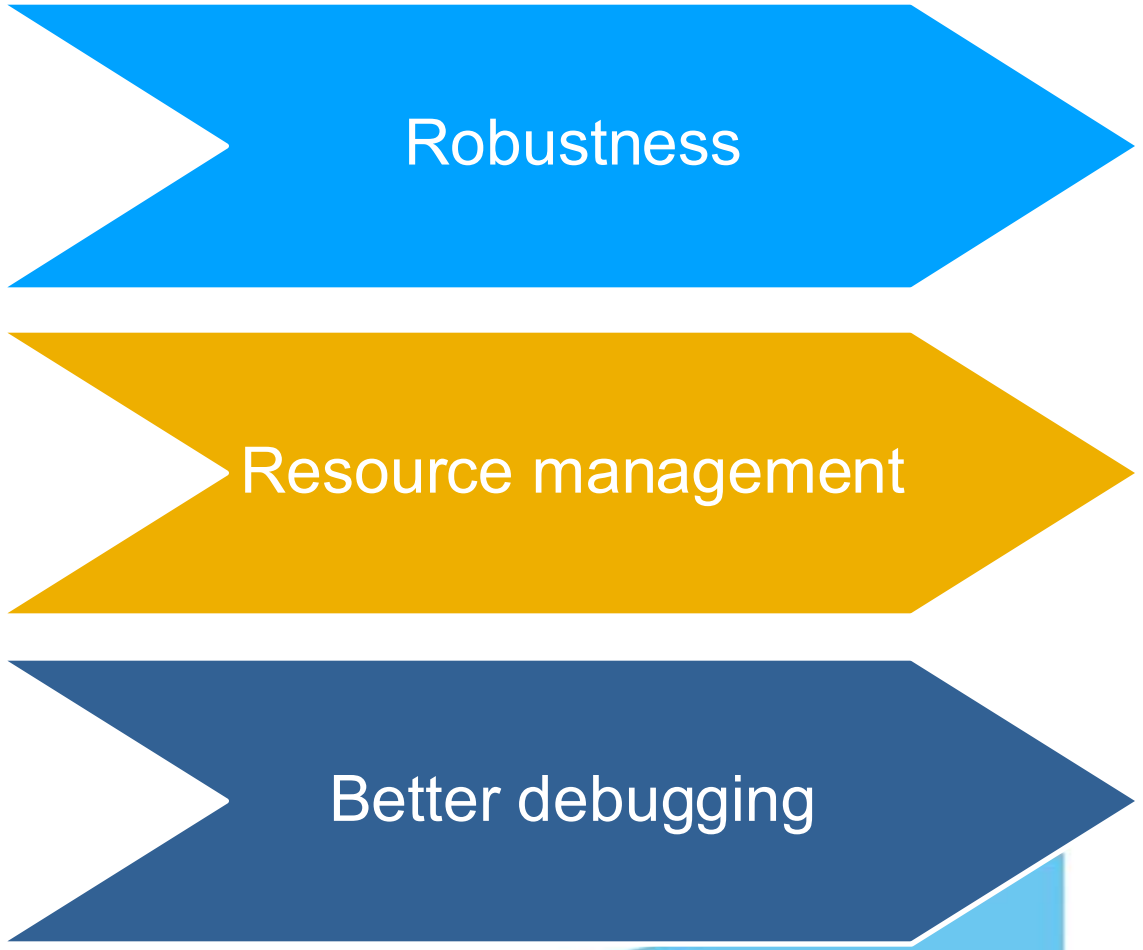

```
def testingDivision(number):
    return(10/number)

try:
    division = testingDivision(int(input("Enter a number")))
except ZeroDivisionError as err:
    print("You can't divide by zero!")
```

A quick reference of exceptions:



Why bother?



Don’t catch them all, be specific!

- Don’t use the except clause without specifying an exception!
- Why? It can hide bugs.
- Think about what exceptions your code could raise!

When everything goes right

- The else clause. Yes, similar to decision structures:

```
try:
    Code to be tested will be placed here
except error:
    Code to let the user know what happened and the options go here.
else:
    If no exceptions were found, execute this code.
```

When everything goes right

```
def testingDivision(number):
    return(10/number)

try:
    division = testingDivision(int(input("Enter a number")))
except ZeroDivisionError as err:
    print("You can't divide by zero!")
else:
    print(division)
```

Cleaning up the mess

- The finally clause:

```
try:
    Code to be tested will be placed here
except error:
    Code to let the user know what happened and the options go here.
else:
    If no exceptions were found, execute this code.
finally:
    code use to release resources (closing files, network connections and database sessions)
```

Cleaning up the mess

```
try:
    file = open('haikai.txt', 'r')
    # do something with the file
except FileNotFoundError:
    print("Your file couldn't be found.")
finally:
    # This will always run, even if an exception occurs above.
    if file:
        file.close()
    print("Cleanup complete.")
```

Raising your own error

- Sometimes you need to flag possible error in your own code.
- One great use is to validate the user input before processing the code!

```
def processingGrades(studentID, grade, module):
    if grade<0:
        raise ValueError("Grade cannot be negative")
    elif isinstance(studentID, int)==False:
        raise ValueError("StudentID should be numerical")
    elif isinstance(module, str) ==False:
        raise ValueError("Module should be alphanumerical")
    else:
        return([studentID, grade, module])

processingGrades("10", 10, "test")
```

```
try:
    grades = processingGrades(""
                              , 10
                              , "test")
except ValueError as e:
    print(f"Error: {e}")
else:
    print(grades)
```

Creating your own exceptions

- Creating exceptions can be a useful way to make your code clear.

```
class notTriangleException(Exception):
    """Exception raised a triangle is not a triangle."""
    pass


def can_make_triangle(a, b, c):
    # Check if all sides are positive
    if a <= 0 or b <= 0 or c <= 0:
        raise ValueError("Integer must be positives")

    # Apply triangle inequality theorem
    if (a + b > c) and (a + c > b) and (b + c > a) == False:
        raise notTriangleException("This is not a possible triangle")
    else:
        return True
```

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Exercises

Exercise

- Use np.zeros to create an array with 12 rows and three columns. Print out the array.
- Next use np.reshape to convert it to an array with 6 rows and 6 columns
- Make a copy of your array and save it to a file.

That’s all folks!