The UK's European university

University of Kent

COMP8270 / PROGRAMMING FOR ARTIFICIAL INTELLIGENCE

Fernando Otero
febo@kent.ac.uk
cs.kent.ac.uk/people/staff/febo

overview:

1. Exceptions

2. NumPy

overview:

1. Exceptions
2. NumPy

Exceptions:

- Handling errors is an important part of building robust programs

```
input = 'something'
float(input)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_4091831/593670237.py in <module>
----> 1 float(a)
ValueError: could not convert string to float: 'something'
```

error type

- Similar to Java, `try/except` block can be used to specify what happens in this cases

Exceptions:

```
def attempt_float(x):
    try:
        return float(x)
    except ValueError:
        return x

attempt_float('1.2345')    # 1.2345
attempt_float('something') # 'something'
atempt_float((1.0, 2.0))   # ?
```

Exceptions:

- Statements can raise different exceptions…

```
attempt_float((1.0, 2.0))


---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_4091831/3808880715.py in <module>
----> 1 attempt_float((1.0, 2.0))

/tmp/ipykernel_4091831/1063157278.py in attempt_float(x)
      1 def attempt_float(x):
      2     try:
----> 3         return float(x)
      4     except ValueError:
      5         return x

TypeError: float() argument must be a string or a number, not 'tuple'
```

Exceptions:

- … and we can capture different exceptions!

```
def attempt_float(x):
    try:
        return float(x)
    except (ValueError, TypeError):
        return x

attempt_float('1.2345')    # 1.2345
attempt_float('something') # 'something'
atempt_float((1.0, 2.0))   # (1.0, 2.0)
```

Exceptions:

- Different behaviour depending on the type of exception

```python
def attempt_float(x):
    try:
        return float(x)
    except ValueError:
        # handle ValueError
    except TypeError:
        # handle TypeError
```

Exceptions:

- Captures all exceptions

```
def attempt_float(x):
    try:
        return float(x)
    except:
        # handle any Exception
```

Exceptions:

- Full notation:

```python
def attempt_float(x):
    try:
        return float(x)
    except:
        # handle any Exception
    else:
        # if no exception, executes this block
    finally:
        # regardless of exception or not, executes
        # this block
```

Example:

```
f = open(path, 'w')

try:
    write_to_file(f)
except:
    print('Failed')
else:
    print('Succeeded')
finally:
    f.close()
```

- The file handle `f` will always get closed

Useful to know:

- Use-defined exceptions: subclasses of `Exception`

- You force an exception to occur:

```
raise Exception("Invalid value for my code")


---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
/tmp/ipykernel_4104649/1911860569.py in <module>
----> 1 raise Exception("Invalid value for my code")

Exception: Invalid value for my code
```

overview:

1. Exceptions

2. NumPy

NumPy:

- Numerical Python is one of the most important foundational packages for numerical computing in Python

- Mostly used for:
  - manipulation of multidimensional arrays (sorting, filtering, transformation, etc.)
  - perform mathematical functions on arrays without having to write loops
  - reading/writing array data to disk

- Why? Because it is **fast**!

NumPy is fast:

- Data is stored in a contiguous block of memory

- NumPy operations (algorithms) are written inC without any type checking or other overhead

- NumPy arrays is much less memory than lists

Try this:

- Generally 10 to 100 times faster (or more) than their pure Python equivalent, using significantly less memory

```
import numpy as np


my_arr = np.arange(1000000)     # numpy array
my_list = list(range(1000000)) # python list


# multiplies every element by 2, repeats 10 times
%time for _ in range(10): my_arr2 = my_arr * 2
%time for _ in range(10): my_list2 = [x * 2 for x in my_list]
```

ndarray:

- ndarray – a n-dimensional **array** object
  - homogeneous data (all elements must be the same type)

package namespace

```
import numpy as np

# creating an array from list
datai = np.array([1, 2, 3])         # int
dataf = np.array([1.0, 2.0, 3.0]) # float
```

- The type is inferred

ndarray:

- Other ways to create ndarrays

```
import numpy as np

data0 = np.zeros(10)    # array with 10 zeros
data1 = np.ones(10)     # array with 10 ones
empty = np.empty(10)    # allocates 10 (float) spaces
data = np.arange(10)    # array from 0 to 9

data = np.array([1, 2.0, 3])  # ?

data = np.array([1, 2, 3])
data[0] = 6.9
print(data)                   # ?
```

multidimensional ndarray:

```
import numpy as np

data3x2 = np.zeros((5, 2))    # 5 (rows) x 2 (columns)
data = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])

data = np.arange(10)          # linear
data = data.reshape((5, 2))  # 5 (rows) x 2 (columns)

print(data.shape)
# (5, 2)
row, columns = data.shape
```

Arithmetic with Arrays:

- Apply operations on data without writing any for loops

- Any arithmetic operations between equal-size arrays applies the operation element-wise

```
import numpy as np

data = np.arange(5)  # [0, 1, 2, 3, 4]
data * data          # [0, 1, 4, 9, 16]
```

Arithmetic with Arrays:

- Arithmetic operations with scalars propagate the scalar argument to each element in the array

```
import numpy as np


data = np.arange(1, 6)  # [1, 2, 3, 4, 5]
data * 5                # [1, 4, 9, 16, 25]
1 / data                # [1.0, 0.5, 0.33, 0.25, 0.2]
```

- **Note:**

```
data = [1] * 5          # ?
```

Statistics:

- NumPy provides functions to compute statistics about an entire array

```
import numpy as np


data = np.random.randn(10)  # 10 random numbers' array
data.mean()                 # or np.mean(data)
data.sum()                  # sum of all elements
```

Statistics:

- You can also specify a particular axis:

```
import numpy as np


data = np.random.randn(2, 5)  # 2 x 5 random array
data.max()                    # whole array
data.max(axis=0)              # per column
data.max(axis=1)              # per row
```

Next lecture:

- Indexing and Slicing NumPy arrays