Intermediate Python

Instructors:
Dmitry Nikolaenko
Jiada Tu

Advanced Research Computing
Durham University

21st February, 2025

0. Python Setup

- In this lesson, we will be using Python 3
- As an interactive computing environment to run Python exercises, there are several options:
  - a web-based interactive environment: *jupyter notebook*
  - the standard interactive shell: *python*
  - an enhanced interactive shell: *ipython*

```
dmitry@dmitry-Latitude-5430:~$ ipython3
Python 3.8.10 (default, Mar 13 2023, 10:26:41)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.13.0 -- An enhanced Interactive Python. Type '?' for help

In [1]:
```

Outline

Introduction

Part I

- 'Beginners Python' recap (variables, lists, control flow statements, and functions)
- Data structures
- Advanced string manipulation techniques

Part II

- *Pythonic* concepts: *args and **kwargs, comprehensions, conditional expressions, iterators, lambdas
- Introduction to modules
- Brief introduction to classes

Outline

By the end of this course, you should know:

- How to write more efficient and *Pythonic* code using advanced language features.
- How to simplify your code with *comprehensions* and *conditional expressions*.
- How to create flexible functions with *variable length* argument list (`*args` and `**kwargs` magic variables).
- How to write and utilize *lambdas* (anonymous functions).
- How to perform advanced *string manipulation* techniques.
- How to import and use *modules* to extend your programs.
- How to work with various data structures and understand the concept of *immutability*.
- How to create and use basic *classes* to implement object-oriented programming principles.

1. Recap: Data types, variable, operators

- **Variables**: a name for a value (*[A-Za-z0-9_]*, case-sensitive)
  - *# this is a comment*
- Basic **data types**: integers, floating-point numbers, strings, bool
    - *pi = 3.14, n = 5, name = “John”*
    - *n = n + 1, n += 1*
    - *str(37)*, *float(‘3.14’)*, *int(pi)*
  - Expressions and operators: +, -, *, /, //, %, **
  - Assignment statements:
    - *degreesC = 5/9*(degreesF - 32)*
    - *w = x = y = z = 0 # chain assignment*
    - *x, y, z = 100, -45, 0 # tuple assignment*

$$^{\circ}C = \frac{5}{9} \times (^{\circ}F - 32)$$

- **Built-in functions**: *type*, *print*, *input*, *eval*, *int*, *float*, *str*, *len*, *range*, ...
    - *print(type(str(1.234)))*
    - *value1 = eval(input('Please enter a number: '))*

1. Recap: Comments, user input, reading/writing files

TBD

1. Recap: Control Flows (repetitions and conditions)

- Conditional statements: *if*, *elif*, *else*

```
grade = eval(input( ' Enter your score: ' ))
if grade>=90:
    print('A')
elif grade>=80:
    print('B')
else:
    print('C')
```

- Loop control statements: *break*, *continue*

```
a = ['Mary', 'had', 'a', 'little', 'lamb', '.']
for i in range(len(a)+2):
    if(i < len(a)):
        print(i, a[i])
    elif (i == len(a)):
        print("The sentence has ended,")
        continue #break
    else:
        print("The end.")
```

- Loop statements: *for*, while

```
for i in range(10):
    print(i)
```

```
i=0
while i<10:
    print(i)
    i=i+1
```

- Nested loops

```
for x_axis in range(2):
    for y_axis in range(2):
        print('(' + str(x_axis) + ',' + str(y_axis) + ')')
```

- Range objects:

range(stop), range(start, stop[, step])

```
range(5)
```

```
range(0, 5)
```

```
list(range(5))
```

```
[0, 1, 2, 3, 4]
```

```
list(range(1,10,2))
```

```
[1, 3, 5, 7, 9]
```

```
len(range(5))
```

```
5
```

```
len(random_stuff)
```

```
4
```

Durham University

1. Recap: Functions

- Principle of encapsulation
  - Enables maintainability and readability alongside complexity
  - Reusable code

Why to write functions:

- It is difficult to write correctly
- It is difficult to debug
- It is difficult to extend

```
def name ( parameter list ):
    block
```

- Important parts of **function** definition:
  - **Name**
  - **Parameters**
  - **Body**
  - Indentation

```python
# function definition
def increment(number):
    number += 1
    new_number = number
    return new_number

# function invocation
print("4+1 =", increment(4))
```

```
4+1 = 5
```

```python
# only function invocation
from math import sqrt
print(sqrt(16.0))
```

```
4.0
```

- Every Python function has two aspects:
  1) Function definition
  2) Function invocation

Durham University

arc

Ask us questions!

2. Data structures: lists (recap)

- **List**: a mutable sequence type
- Holds a collection of objects in a defined order (indexed by integers)
- A user-friendly data container to store objects of any type, even a mix of types
- Dynamic type system:
  - no need to declare the type of a variable explicitly

```
L = [1, 2, 3, 4, 5]
print(type(L))
```

```
<class 'list'>
```

```
random_stuff = [1, 'apples', 3.14, ['Mars', 'Venus', 'Pluto']]
print(random_stuff[3][2])
```

```
Pluto
```

Appending to lists:

```
chem_elements = ["oxygen"]
for i in range(2):
    chem_elements.append("hydrogen")
print(chem_elements)
```

- Easy to process lists in *for* loops (in the next slides)

Durham University

2. Data structures: list methods

- Inserting an element
- Reversing
  - Note: *reversed()* returns a 'reverse iterator' that then needs to be turned back into a list with list()
- Sorting
- Searching
- Emptying
- Removing duplicates
  - by converting list→set→list
  - *Note: As we can see from examples, some functions actively modified whereas some returned a copy of the modified list*

```
L = [33, 84, 11, 29, 0]
L.insert(2, 57)
print(L)
```

```
[33, 84, 57, 11, 29, 0]
```

```
print(list(reversed(L)))
```

```
[0, 29, 11, 57, 84, 33]
```

```
print(sorted(L))
```

```
[0, 11, 29, 33, 57, 84]
```

```
if 57 in L:
    position = L.index(57)
    print("57 is in the list. "
          "It is at position", position)
```

```
57 is in the list. It is at position 2
```

```
print(L.clear())
```

```
None
```

```
L2 = [1, 2, 2, 2, 3, 3]
print(list(set(L2)))
```

```
[1, 2, 3]
```

2. Data structures: similarities of lists to strings

- *len* function: the number of items in a list/string
- *in* operator: tells if a list/string contains something
- + and * operators: concatenating and repeating

```
[7,8]+[3,4,5]
```

```
[7, 8, 3, 4, 5]
```

```
[0]*5
```

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

- **Indexing**: simple to "grab" an item/character in a list/string if you know where it sits:

```
L = ['a','b','c','d','e','f','g','h','i','j']
print(L[4])
```

```
e
```

```
L = "abcdefghij"
print(L[4])
```

```
e
```

The :stop value represents the first value not in the slice

- **Slicing**: use : to "grab" a range defined subsection of a list/string:

```
start=3
stop=7
```

```
print(L[start:stop]) # items start to stop-1
```

```
['d', 'e', 'f', 'g']
```

```
print(L[start:]) # items start to the end of list
```

```
['d', 'e', 'f', 'g', 'h', 'i', 'j']
```

```
print(L[:stop]) # items from beginning of list to stop-1
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

```
print(L[:]) # whole list
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

- **Looping**:

```
for i in range(len(L)):
    print(L[i])
```

```
for item in L:
    print(item)
```

2. Data structures: dictionaries

- **Dictionaries**: flexible mappings of keys to values
  - an unordered collection of key-value pairs
  - dictionary items are colon-connected (:) key-value pairs
  - an associative container permitting access based on a key (not an index)

```
capitals = {'Norway':'Oslo','Sweden':'Stockholm','France':'Paris'}
# capitals = dict(Norway='Oslo', Sweden='Stockholm', France='Paris') # the same
capitals['Germany'] = 'Berlin' # instead of append() for list
for country in capitals:
    print(f'The capital of {country} is {capitals[country]}')
```

```
The capital of Norway is Oslo
The capital of Sweden is Stockholm
The capital of France is Paris
The capital of Germany is Berlin
```

2. Data structures: sets

- **Sets**: unordered collections of unique elements
  - represents a mathematical set
  - curly braces ({}) enclose the elements of a literal set

```
S = {10, 3, 7, 2, 11}
S
```

```
{2, 3, 7, 10, 11}
```

```
L = [10, 13, 10, 5, 6, 13, 2, 10, 5]
S = set(L)
L
```

```
[10, 13, 10, 5, 6, 13, 2, 10, 5]
```

```
S
```

```
{2, 5, 6, 10, 13}
```

2. Data structures: tuples

- **Tuples**: immutable sequences
  - essentially a constant list which can't be changed

```
t = (2, 4, 6, 'temp.pdf')
t = 2, 4, 6, 'temp.pdf' # can skip parentheses
t
```

```
(2, 4, 6, 'temp.pdf')
```

  - much of the same functionality as lists, including **indexing** and **slicing**

```
t = t + (-1.0, -2.0)
t
```

```
(2, 4, 6, 'temp.pdf', -1.0, -2.0)
```

```
t[1]
```

```
4
```

```
t[3:]
```

```
('temp.pdf', -1.0, -2.0)
```

```
6 in t
```

```
True
```

  - Lists are typically for homogeneous data sequences (ingredients, names) whereas tuples are ideal for heterogeneous data (entries with different meanings)

2. Data structures: namedtuple

- **Namedtuple** is handy, but also immutable
- Namedtuple is a factory function for making a tuple class
  - In the example, *NINumber* becomes a factory function that can encapsulate data from any employee

```
from collections import namedtuple
NINumber = namedtuple('national_insurance_number', 'name NI')
employee_data = NINumber('Simon', '12345678')
print(f'Employee {employee_data.name} has NI: {employee_data.NI}')
```

```
# assignment creates error!
# employee_data.NI = '4444'
```

Durham University

Exercises

Ask us questions!

3. Advanced string manipulation

- Adjusting case
- Formatting strings
    - *Note: Modification requires assignment, because these functions return a copy, not modifying the original string*

```
line = "the quick brown fox jumped over a lazy dog"
print(line.find('fox'))
print(line.startswith('the'))
print(line.endswith('fox'))
print(line.replace('brown', 'red'))
print(line.split())
try:
    index = line.index('bear')
    print(index)
except ValueError:
    print("A bear isn't mentioned in the text")
```

```
16
True
False
the quick red fox jumped over a lazy dog
['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'a', 'lazy', 'dog']
A bear isn't mentioned in the text
```

```
arc_update = "ThE HAmILton suPercompUTER is beiNg UPGraded"
print(arc_update.upper())
print(arc_update.title())
print(arc_update.capitalize())
print(arc_update)
arc_update = "   RSE   "
print(arc_update.strip())
print(arc_update.rstrip())
print(arc_update.lstrip())
```

```
THE HAMILTON SUPERCOMPUTER IS BEING UPGRADED
The Hamilton Supercomputer Is Being Upgraded
The hamilton supercomputer is being upgraded
ThE HAmILton suPercompUTER is beiNg UPGraded
RSE
   RSE
RSE   
```

- *Find()* and *index()*: return index of a substring, but the latter raises a *ValueError* exception when not found (**exception handling**)
- Querying the existence, replacing, splitting

3. Advanced string manipulation: F-strings

- The canonical way to search a string (if not interested in the index) is very simple:

```
line = "the quick brown fox jumped over a lazy dog"
if "fox" in line:
    print("A fox has been seen")
```

```
A fox has been seen
```

- **F-strings** provide a way to embed expressions inside string literals, using a minimal syntax
  - expressions are evaluated at runtime and replaced with their values

```
interests = ["football", "zoom"]
print(f"Bob enjoys {interests[0]} and {interests[1]}")

weekdays = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
for weekday in weekdays:
    print(f"Today is {weekday}")

age = 70
print(f"Soon I'll be {age+1}!")
```

```
Bob enjoys football and zoom
Today is Mon
Today is Tue
Today is Wed
Today is Thu
Today is Fri
Soon I'll be 71!
```

Exercises

Ask us questions!

4. Pythonic concepts: *iterators* beneath control flows

But what is the *for* loop doing under the hood?

1. *iter()* is called on the container object returning an iterator object
2. The iterator object defines a *__next__()* function which facilitates access to elements one at a time
3. *__next__()* tells for loop when there are no more elements raising *StopIteration* exception

```python
uni = 'Durham'
it = iter(uni)
it
```

```
<str_iterator at 0x7f1208f38f70>
```

```python
next(it)
```

```
'D'
```

```python
next(it)
```

```
'u'
```

```python
next(it)
```

```
'r'
```

```python
next(it)
```

```
'h'
```

```python
next(it)
```

```
'a'
```

```python
next(it)
```

```
'm'
```

```python
next(it)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-31-bc1ab118995a> in <module>
----> 1 next(it)

StopIteration:
```

Durham University

arc

4. Pythonic concepts: *args and **kwargs

- Collecting an arbitrary number of arguments into a tuple, when number of arguments is unknown, **args*:

```
def product(*nums):
    prod = 1
    for i in nums:
        prod*=i
    return prod
print(product(3,4), product(2,3,4), sep=' ')
```

```
12 24
```

- Collecting an arbitrary number of keyword arguments into a dictionary, ***kwargs*:

```
def f(**kwargs):
    for k in kwargs:
        print(k, '**2 : ', kwargs[k]**2, sep='')
f(x=3, y=4, z=5)
```

```
x**2 : 9
y**2 : 16
z**2 : 25
```

```
def f(**kwargs):
    for key, value in kwargs.items():
        print(key, "=", value, sep='', end=", ")
f(x=3, y=4, z=5)
```

```
x=3, y=4, z=5,
```

Durham University

arc

4. Pythonic concepts: conditional expressions and comprehensions

- Conditional expressions

```
hungry = True
state = "grumpy" if hungry else "content"
print(state)
```

```
grumpy
```

- List comprehension

```
multiples_of_three = [i for i in range(20) if i%3==0]
print(multiples_of_three)
```

```
[0, 3, 6, 9, 12, 15, 18]
```

- List methods (more after exercise)

```
L = [33, 84, 57, 11, 29, 0, 57]
L.remove(57)
print(L)
```

```
[33, 84, 11, 29, 0, 57]
```

```
L.remove(57)
print(L)
```

```
[33, 84, 11, 29, 0]
```

4. Pythonic concepts: lambda functions

- **Lambda functions** for compact inline function definitions
- Useful when you don't want to use a function twice

lambda arguments : expression

Or more generally:

Functionally equivalent

```
somefunc = lambda a1, a2, ...: some_expression
```

```
def somefunc(a1, a2, ...):
    return some_expression
```

- Example

$$f''(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}$$

```
def diff2(f, x, h=1E-6):
    r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
    return r
```

```
def f(x):
    return x**2 - 1

df2 = diff2(f,1.5)
print(df2)
```

equivalent

```
df2 = diff2(lambda x: x**2-1,1.5)
print(df2)
```

```
1.999733711954832
```

Durham University

4. Pythonic concepts: lambda functions (cont.)

- **Map** applies a function to all the items in an iterable:

map(function_to_apply, list_of_inputs)

```
items = [1, 2, 3, 4, 5]
squared = []
for i in items:
    squared.append(i**2)
print(squared)
```

equivalent

```
squared = list(map(lambda x: x**2, items))
print(squared)
```

```
[1, 4, 9, 16, 25]
```

```
squared = [x**2 for x in items]
```

- **Filter** creates a list of elements for which a function returns true:

*Note: **list comprehensions** can accomplish everything what **map** and **filter** do*

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x<0, number_list))
print(less_than_zero)
```

```
def is_less_than_zero(x):
    return True if x<0 else False

negative_nums = []
for num in number_list:
    if is_less_than_zero(num):
        negative_nums.append(num)
print(negative_nums)
```

equivalen
t

```
negative_nums = [num for num in number_list if num < 0]
print(negative_nums)
```

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

Exercises

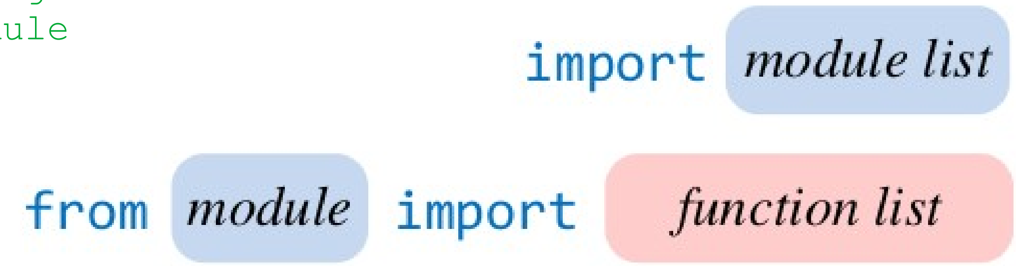
Ask us questions!

5. Modules

- Several ways of importing **modules**:

```
import os, sys, time # several modules at once
import numpy as np # changing module name
import math # the whole module

print(math.sqrt(16.0))
```



- Another example

```
from math import sqrt # only specific functions from a module
print(sqrt(16.0))

from math import * # all functions from a module
from math import log as ln # changing function name
```

5. Modules: some standard and 3rd-party modules

- **math**: contains familiar math functions including:
  - *sin* , *cos* , *tan* , *exp* , *log* , *log10* , *factorial* , *sqrt* , *floor* , *ceil*
- **numpy**: fundamental package for scientific computing

a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more

- **scipy**: a collection of mathematical algorithms and convenience functions built on the NumPy extension

high-level commands and classes for the manipulation and visualization of data

- **matplotlib**: library for plotting
- **sympy**: symbolic computations
- Other useful modules: **os**, **random**, **itertools**, **time**, **datetime** and many more
- Python comes with a program called *pip* which will automatically fetch packages released and listed on PyPI: *pip install <some-module>*

Durham University

5. Modules: other useful modules in the standard library

- Tools for interfacing with the operating system: **os**

Facilitates portability

```
import os
root = "/Users"
print(os.path.join(root, os.environ["USER"], "holiday_planning"))
# out  /Users/kqkc25/holiday_planning

os.listdir("Desktop")

if not os.path.exists("blahblahblah.txt"):
    print("File not found")
    exit(1)
```

Very useful when processing multiple data files

5. Modules: using the *csv* module (part 1)

- Very convenient module for parsing and writing csv files

- Writing a csv

```
import csv

with open("example.csv", "w") as out_f:
    writer = csv.writer(out_f, delimiter=",")
    writer.writerow(["x_axis", "y_axis"])
    x_axis = [x * 0.1 for x in range(0, 100)]
    for x in x_axis:
        writer.writerow([x, math.cos(x)])
```

For the sake of visualization, here is the first part of the csv we just made:

x_axis	y_axis
0	1
0.1	0.99500417
0.2	0.98006658
0.3	0.95533649
0.4	0.92106099
0.5	0.87758256
0.6	0.82533561
0.7	0.76484219
0.8	0.69670671
0.9	0.62160997
1	0.54030231
1.1	0.45359612
1.2	0.36235775
1.3	0.26749883
1.4	0.16996714
1.5	0.0707372

5. Modules: using the *csv* module (part 2)

- Now let's extract the value for y_axis when x_axis is 1.0 for the csv we just wrote:

```
import csv

with open("example.csv", "r") as in_file:
    reader = csv.reader(in_file, delimiter=",")
    next(reader)  # skip header
    for row in reader:
        if row[0] == "1.0":
            print(row[1])
            break

# out:  0.5403023058681398
```

Exercises

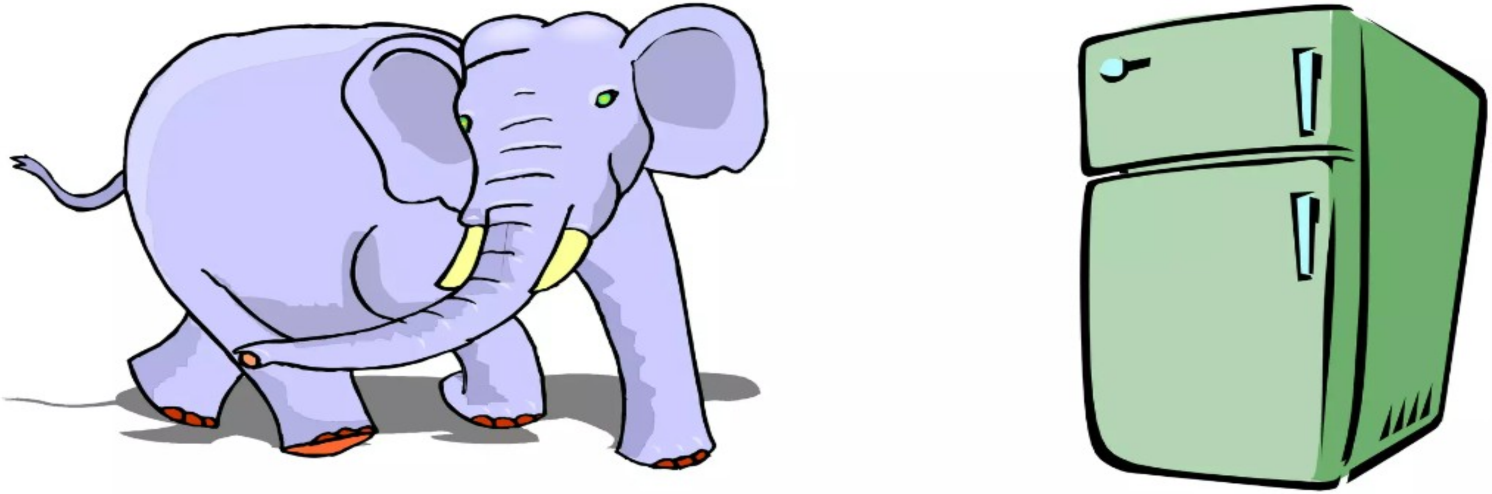
Ask us questions!

6. Brief introduction to classes

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.

What is OOP?

Q: How do you put an elephant into a refrigerator?



```python
# define element size
elephant_x = 5
elephant_y = 5
elephamt_h = 5

# define refrigerator size
refrigerator_x = 6
refrigerator_y = 6
refrigerator_h = 6


def open_refrigerator_door():
    print('Refrigerator door is opened')

def package_elephant():
    print('Elephant is packaged')

def put_elephant_to_refrigerator():
    print('Elephant is in the fridge already')

def close_refrigerator_door():
    print('Close the refrigerator door')


# compare size
if elephant_x < refrigerator_x and elephant_y < refrigerator_y and elephamt_h < refrigerator_h:
   open_refrigerator_door()
   package_elephant()
   put_elephant_to_refrigerator()
   close_refrigerator_door()
else:
    print('refrigerator is too small to put elephant')
```

Durham University

```python
class Elephant:
    def __init__(self, x, y, h):
        self.x = x
        self.y = y
        self.h = h

    def package(self):
        print('Elephant is packaged')

class Refrigerator:
    def __init__(self, x, y, h):
        self.x = x
        self.y = y
        self.h = h
        self.is_door_open = False

    def open_door(self):
        self.is_door_open = True
        print('Refrigerator door is opened')

    def close_door(self):
        self.is_door_open = False
        print('Close the refrigerator door')

    def put_elephant(self, elephant):
        if not self.is_door_open:
            self.open_door()
        if elephant.x < self.x and elephant.y < self.y and elephant.h < self.h:
            elephant.package()
            print('Elephant is in the fridge already')
        else:
            print('Refrigerator is too small to put elephant')
        self.close_door()
```

```python
# Define element sizes
elephant_x = 5
elephant_y = 5
elephant_h = 5

# Define refrigerator sizes
refrigerator_x = 6
refrigerator_y = 6
refrigerator_h = 6

# Create instances of Elephant and Refrigerator classes
elephant = Elephant(elephant_x, elephant_y, elephant_h)
refrigerator = Refrigerator(refrigerator_x, refrigerator_y, refrigerator_h)

# Put the elephant in the refrigerator
refrigerator.put_elephant(elephant)
```

Durham University

arc

Exercise

Attributes:

- make: The make of the car (e.g., Toyota, Honda).
- model: The model of the car (e.g., Camry, Civic).
- year: The year the car was manufactured.
- color: The color of the car.
- mileage: The current mileage of the car.

Methods:

- __init__: The constructor method to initialize the attributes.
- drive(distance): A method that takes a distance in miles as a parameter and increases the mileage of the car accordingly.
- paint(new_color): A method that changes the color of the car.
- display_info(): A method that displays all the information about the car (make, model, year, color, and mileage).

Exercises

Ask us questions!

Durham University

Thank you!

– Feedback would really be appreciated: **https://bit.ly/arc_trainingfeedback2024**
– Other training courses at ARC
– RSE support