Lecture 3.1

Functions

Topic 3.1 and 3.2 Intended Learning Outcomes

- By the end of the week you should be able to:
  - Define and call functions to reuse sections of code,
  - Define custom types of objects with classes, and
  - Understand how references work, and the difference between mutable and immutable objects.

Lecture Overview

1. Function Calls
2. Writing Functions
3. Parameters, Scope, and Return Values

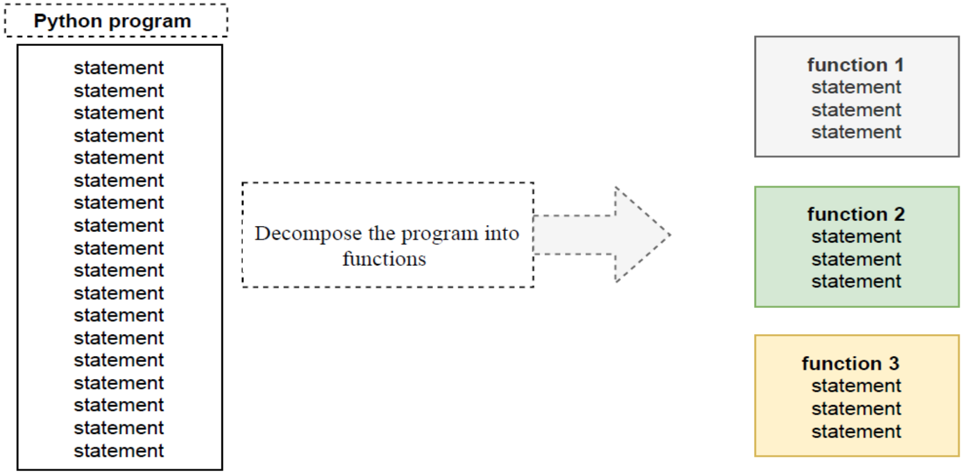
Function

What is a Function?

A function is a **sequence** of **statements** with a **name**.

- Good functions are self-contained and designed to accomplish a **specific** task.
- Because a function can be identified by its **name**, we can call the function to perform the task when we need to.
- Using functions **reduces** the amount of repeated code and makes programming more **accessible**.
- As program grows larger and larger, functions make it more **organised** and **manageable**.

What is a Function?

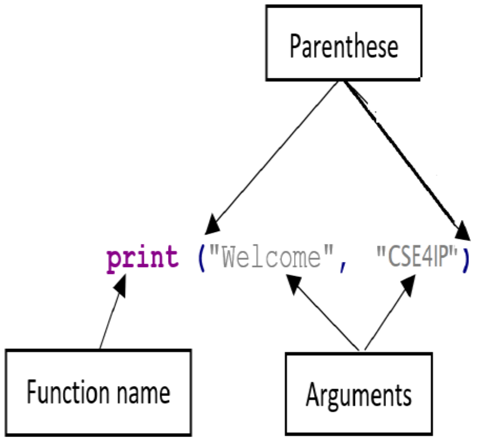


- Task decomposition: They allow us to divide a task into smaller sub-tasks.
- Program structuring: They allow us to make our programs well- structured with meaningful components and clear relationships among them.
- Other advantages include: simpler and easier to understand, code reuse, very helpful when developing large program by teams.

What is a Function?

An example of function that we have used before.

```
print ("Welcome", "CSE4IP")
```

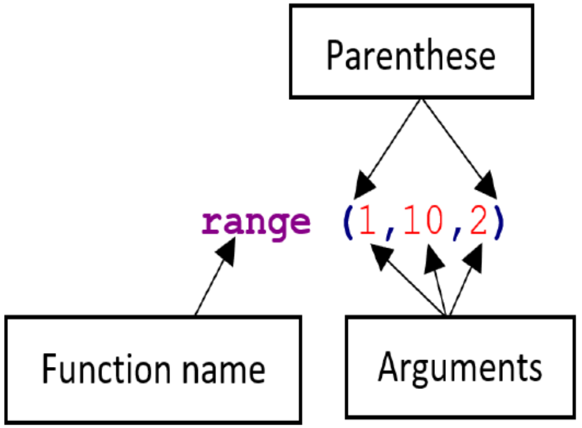


- `print` is the function name, not an argument.
- `"Welcome"` is the first argument.
- `"CSE4IP"` is the second argument.
- Any kind of **expression** can be used as an argument.

What is a Function?

An example of function that we have used before.

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



- `range` is the function name, not an argument.
- `1` is the first argument.
- `10` is the second argument.
- `2` is the third argument.

Function Calls

- To **run** the code in a **function**, you **must** **call** the **function**.

- We have already been using function calls! Here are some that should look familiar:
  - `print('Hello')`
  - `input('Enter your age: ‘)`
  - `range (1,6)`

Arguments

- The inputs supplied to a function are called arguments.

- Any kind of **expression** can be used as an **argument**:
  - Literals (e.g. `type(65.3)`),
  - Variables (e.g. `str(temperature)`), or
  - Complex expressions (e.g. `print('$' + str(x * 2))`).

- **Multiple** arguments are **separated** using **commas**.

Things Functions Can Do

- A function can cause an effect.
  - For example, the print function **causes** an **output** message to be **displayed**.

- A function can compute and return a result.
  - For example, the str function* (`str(x * 2)`) **converts** its **argument** to a string and **returns** the result.

Check Your Understanding

Q. What is the second argument in this function call?

```
print('I, Jimbo, am', 18, 'years old')
```

Check Your Understanding

Q. What is the second argument in this function call?

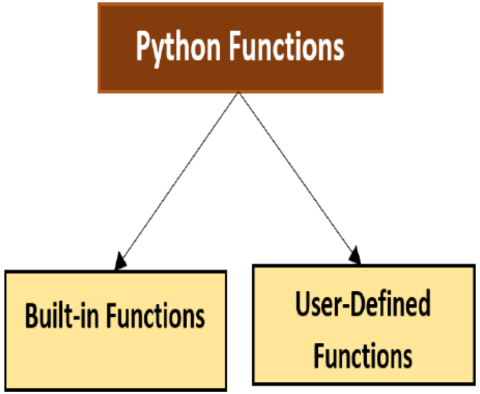
```
print('I, Jimbo, am', 18, 'years old')
```

A. 18 is the second argument.

- print is the function name, not an argument.
- 'I, Jimbo, am' is the first argument.
  - These commas are part of the string.
- 'years old' is the third argument.

Pythom Functions

— **Python functions can be classified into two types as follows:** —



1. **Built-in Functions**: available in Python by default, e.g., **print**, **range**.
2. **User-Defined Functions**: defined and written by programmers (ourselves).

Built-in Functions

Built-in functions

Built-in functions are set of functions implemented within Python interpreter. These functions readily available for use by user. You just need to provide the input and the function will return the output.

		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Source: https://docs.python.org/3/library/functions.html

```
import builtins
print (dir(builtins))
```

Built-in Functions

Examples of Built-in Functions

— abs() method returns the absolute value of the given number

```
>>> integer = -10
>>> integer_abs = abs(integer)
>>> print("Absolute value of -10 is:", integer_abs)
Absolute value of -10 is: 10
```

— round() rounds the number into the nearest integer

```
>>> round(1.99)
2
>>> round (1.2)
1
```

— len() returns the number of elements in iterable object.

```
>>> s = "Welcome to CSE4IP"
>>> l = len(s)
>>> print("The number of elements is:", l)
The number of elements is: 17
```

Built-in Functions

Examples of Built-in Functions

— min() returns the smallest element in two or more parameters.

```
>>> m = min(1, 3, 2, 5, 4)
>>> print("The minimum is:", m)
The minimum is: 1
```

— pow() function returns the power of a number.

```
>>> x = 4
>>> r = pow(x,2)
>>> print("The power of $x^2$ is:", r)
The power of $x^2$ is: 16
```

— max() returns the bigest element in two or more parameters.

```
>>> m = max(1, 3, 2, 5, 4)
>>> print("The maximum is:", m)
The maximum is: 5
```

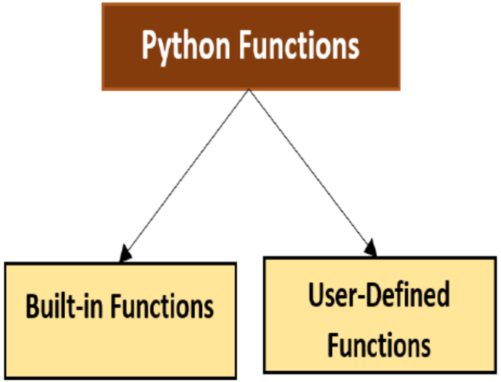
— eval() evaluats the input as a Python expression

```
>>> eval('2 * 3 + 5.6')
11.6
```

Writing Functions - User-Defined Functions

Pythom Functions: User-Defined Functions

— Python functions can be classified into two types as follows: —



1. **Built-in Functions**: available in Python by default, e.g., **print**, **range**.
2. **User-Defined Functions**: defined and written by programmers (ourselves).

Writing a Function: User-Defined Functions

A function is a **sequence** of **statements** with a **name**.

— What are User-Defined Functions? —

User-Defined Functions

User-Defined Functions are defined and written by programmer (ourselves) to do a specific task(s). Functions can be **identified** by their **names**.

Once a function is defined, we can call the function to perform the task when we need to.

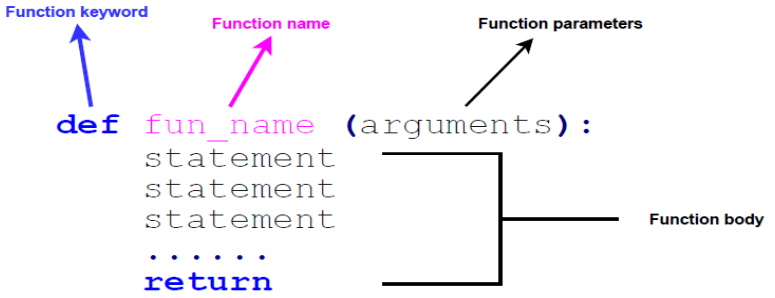
- Functions that readily come with Python are known as **built-in functions**.
- If we call functions written by others in the form of the library, this known as **library functions (module)**.
- A user-defined function could be a **library function** to someone else.
- All the other functions that we **write** are known as **user-defined functions**.

Writing a Function: User-Defined Functions

How to define a function?

Pseudo Code: Python function

```
def fun_name (arguments):
    statement
    statement
    return
```



The function is defined with the def keyword followed by function name, parentheses and ends with a colon. The statements that are part of the function are indented below the def keyword.

Writing a Function: User-Defined Functions

Example: Write a function to print `"Welcome to CSE4IP"`

Example: Defining a function syntax

```
def my_function():
    print("Welcome to CSE4IP")
```

- **my_function()** is the function name.
- The function does not accept any parameters as inputs i.e., empty parentheses.
- The function has one statement only: `print("Welcome to CSE4IP")`
- The statement is indented below the **def** keyword.
- The function will display `"Welcome to CSE4IP"` on the screen.
- The function did not **return** the result.

Writing a Function: User-Defined Functions

Calling a Function

A function definition specifies function tasks, but it does not execute function statements (code). To execute a function code, we must call it within our program. This example demonstrates how we would call the **my_function()** function:

Example: Calling a function syntax

```
def my_function():
    print("Welcome to CSE4IP")

# Call the defined function
my_function()

# Function output
>>> "Welcome to CSE4IP"
```

Writing a Function: Control Flow

- When a function is called, **control** flow **jumps** to the **first** statement in the **function**.

- When the function finishes, **control** flow is **returned** to the **place** that the function was **called** from.

- The statements in a function are *not* **executed** when the function is **defined**.

Example: Control Flow

```python
def tick():
    print('Tick')
    print('Tock')
print('Who am I?')
tick()
print('I am a clock')
tick()
print('Goodbye!')
```

- Let's look at the control flow for this program.
- The statements within the `tick` function are executed with each function call.

Example: Control Flow

```
def tick():
    print('Tick')
    print('Tock')
print('Who am I?')
tick()
print('I am a clock')
tick()
print('Goodbye!')
```

Output:

Example: Control Flow

```python
def tick():
    print('Tick')
    print('Tock')
print('Who am I?')
tick()
print('I am a clock')
tick()
print('Goodbye!')
```

Output:

```
Who am I?
```

Example: Control Flow

```python
def tick():
    print('Tick')
    print('Tock')
print('Who am I?')
tick()
print('I am a clock')
tick()
print('Goodbye!')
```

Output:

```
Who am I?
```

Example: Control Flow

```
def tick():
    print('Tick')
    print('Tock')
print('Who am I?')
tick()
print('I am a clock')
tick()
print('Goodbye!')
```

Output:

```
Who am I?
Tick
```

Example: Control Flow

```python
def tick():
    print('Tick')
    print('Tock')
print('Who am I?')
tick()
print('I am a clock')
tick()

print('Goodbye!')
```

Output:

```
Who am I?
Tick
Tock
```

Example: Control Flow

```python
def tick():
    print('Tick')
    print('Tock')
print('Who am I?')
tick()
print('I am a clock')
tick()
print('Goodbye!')
```

Output:

```
Who am I?
Tick
Tock
I am a clock
```

Example: Control Flow

```
def tick():
    print('Tick')
    print('Tock')
print('Who am I?')
tick()
print('I am a clock')
tick()
print('Goodbye!')
```

Output:

```
Who am I?
Tick
Tock
I am a clock
```

Example: Control Flow

```
def tick():
    print('Tick')
    print('Tock')
print('Who am I?')
tick()
print('I am a clock')
tick()
print('Goodbye!')
```

Output:

```
Who am I?
Tick
Tock
I am a clock
Tick
```

Example: Control Flow

```
def tick():
    print('Tick')
    print('Tock')
print('Who am I?')
tick()
print('I am a clock')
tick()
print('Goodbye!')
```

Output:

```
Who am I?
Tick
Tock
I am a clock
Tick
Tock
```

Example: Control Flow

```
def tick():
    print('Tick')
    print('Tock')
print('Who am I?')
tick()
print('I am a clock')
tick()
print('Goodbye!')
```

Output:

```
Who am I?
Tick
Tock
I am a clock
Tick
Tock
Goodbye!
```

Don't Repeat Yourself

- In software development, there's a good practice called DRY (**d**on't **r**epeat **y**ourself).
- DRY code avoids code duplication (writing similar code multiple times).
- Functions are an effective tool for producing DRY code.

Example: Code Duplication

```python
1 water = 0.712
2 print('Covered in water:')
3 print(str(water * 100) + '%')
4 land = 1 - water
5 print('Covered in land:')
6 print(str(land * 100) + '%')
```

- Lines 3 and 6 have the same purpose: to display a percentage.
- Problems:
  - It is cumbersome to copy/paste.
  - Changing the way percentages are displayed requires multiple edits.
- This is error-prone.

Example: Code Duplication

```python
water = 0.712
print('Covered in water:')
print(str(round(water*100))+'%')
land = 1 - water
print('Covered in land:')
print(str(round(land*100))+'%')
```

- Here we had to edit 2 lines to print percentages as round numbers.
- In a larger program, many more edits could be required.
  - It's easy to forget one, or to make a mistake.

Example: DRY Code with a Function

```
1 def print_percent(x):
2   print(str(round(x*100))+'%')
3
4 water = 0.712
5 print('Covered in water:')
6 print_percent(water)
7 land = 1 - water
8 print('Covered in land:')
9 print_percent(land)
```

- Lines 1--2 define the `print_percent` function.
  - The function has one parameter, `x`.
- Lines 6 and 9 call the function.
  - The code in `print_percent` will be executed with `x` set to the argument value.

Check Your Understanding

Q. How many times will the shown program print output?

```python
def funky_func():
    for i in range(3):
        print('Groovy!')
funky_func()
if 2 + 2 == 5:
    funky_func()
funky_func()
```

Check Your Understanding

Q. How many times will the shown program print output?

A. 6 times.

- Each time funky_func is called, 'Groovy!' is printed 3 times.
- funky_func is called twice (lines 4 and 7).
  - Not called from line 6 since the if statement condition is false.

```python
1 def funky_func():
2     for i in range(3):
3         print('Groovy!')
4 funky_func()
5 if 2 + 2 == 5:
6     funky_func()
7 funky_func()
```

Parameters, Scope, and Return Values

Parameters and Arguments.

- **Parameters** are function input variables declared as part of defining a function.
  - A function can have **zero**, **one**, or **many** parameters.
  - Act like "**placeholders**".

- **Arguments** are the values assigned to parameters when calling a function.
  - A function **must** receive **one** argument **per parameter**.

Example: Parameters and Arguments

```
1 def print_product(x, y):
2     print(x * y)
3
4 print_product(3, 5)
```

- **x** and **y** are **parameters**.
- Line 1: "define a function called print_product with two parameters, **x** and **y**".

Example: Parameters and Arguments

```
1 def print_product(x, y):
2     print(x * y)
3
4 print_product(3, 5)
```

- **3** and **5** are arguments.
- Line 4: "call print_product with **3** as the **first argument** and **5** as the **second** argument".
- The order matters---**x** will take the value of **3**, and **y** will take the value of **5**.

Function Default Arguments

- A default argument is a value given to a **parameter** when **no** argument is explicitly provided in the function call.

- You can specify **default arguments** as part of the function **definition**.

Function Default Arguments

Function Arguments: Default Parameter Value

A function can be called without parameter. It will use the default value.

Default Parameter Value

```
>>> def my_function(country = "Australia"):
...     print("I am from " + country)
...
>>> my_function("India")
I am from India
>>> my_function("China")
I am from China
>>> my_function()
I am from Australia
>>> my_function("UK")
I am from UK
```

Default Parameter Value

```
>>> def print_multiple(string, n=1):
...     print(string * n )
...     print()
...
>>> print_multiple('CSE4IP')
CSE4IP

>>> print_multiple('CSE4IP', 4)
CSE4IPCSE4IPCSE4IPCSE4IP
```

Example: Default Arguments

```python
def print_ticket(age, max_child_age=12):
    if age <= max_child_age:
        print('Child')
    else:
        print('Full fare')

print_ticket(5)  #=> Child
print_ticket(15)  #=> Full fare
print_ticket(15, 18)  #=> Child
```

Default Arguments

Beware!

Once you specify a **default argument** for one parameter, **all** parameters which come **after it must** also have **default** arguments.

```
>>> def do_something(a, b=1, c):
...     pass
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

Named Arguments

- Arguments can also be specified by **name**.
- These kinds of arguments are called named arguments (or keyword arguments).
- The order of named arguments **doesn't matter**.
- All **named** arguments **must** appear **after** **positional** arguments.

```python
def divide(num, denom):
    print(num / denom)

divide(5, 2) #=> 2.5
divide(denom=2, num=5) #=> 2.5
divide(5, denom=2) #=> 2.5
divide(num=5, 2) #=> Error
```

Named Arguments

- In most cases you will **not *need*** to use named arguments.
- If you think that they will make your code more **understandable**, go ahead and use them.
- Occasionally you will **encounter** functions that do actually **require** named **arguments**.
  - We won't see many in this subject.
  - It will be pointed out to you when we do.

Example: Variable Scope

```python
def double_number(x):
    y = x * 2

double_number(5)
print(y)
```

```
Traceback (most recent call last):
  File "scope.py", line 5, in <module>
    print(y)
NameError: name 'y' is not defined
```

- The code outside of double_number can't "see" **x** and **y**.
  - It's as if those variables don't exist.
- As a result, the line containing the print statement will raise an error.

Return Values

- If **variables** created **inside** a function **aren't accessible outside** it, then how do we **get** the result of a **calculation** performed **inside** the function?
    - Answer: use a **return** value.
    - A function can return one or more values using the return statement.

Return Values.

- A function can have a return value, which is a value from inside the function which is returned to the caller of the function.
- The function **call** will **evaluate** to the **returned** value.
- We've actually already used several functions with return values.
  - e.g. `abs(-3)` **evaluates** to **3** (its return value).
- We can **specify** which **value** is **returned** from a function by using a return statement.

Return Statements

- A return statement can **only** be used **inside** a function.

- In Python, a **return** statement **consists** of the return **keyword followed** by a **value** to be **returned**.
  - `return 'A cool result'`
  - `return x + 1`

- When a **return** statement is **encountered**, function execution **finishes** and the specified **value** is **returned** to the **caller**.

Example: Return Statement

```
def double_number(x):
    y = x * 2
    return y

z = double_number(5)
print(z)
```

- The value of variable **y** is returned from function **double_number**.
- This means that the expression double_number(5) will **evaluate** to **10**.

Example: Return Statement

Return Values

```
>>> def my_function(value):
...     return 6 * value
...
>>> print(my_function(3))
18
>>> x= my_function (4)
>>> print(x)
24
>>> print(my_function(x))
144
```

Return Values

```
>>> def my_function(value):
...     x=value+1
...     y= 6 * value
...     return x,y
...
>>> print(my_function(3))
(4, 18)
>>> x = my_function(3)
>>> print (x)
(4, 18)
>>> print (type(x))
<class 'tuple'>
```

Finishing Function Execution Early

- After a return statement is encountered, the function **finishes** immediately.
  - Statements which appear **after** the return statement will **not be executed**.
  - This is similar to continue and break in loops.

- The return keyword can be **used** by itself to **finish** a function **without** also **returning a value**.

Check Your Understanding

Q. What is the output of the shown program?

```python
def do_something(n):
    if n < 0:
        return 0
    return n ** 2

print(do_something(-4))
```

Check Your Understanding

Q. What is the output of the shown program?

A. 0.

- do_something is called with **-4** as the first argument (so **n** is **-4**).
- -4 is less than 0, so 0 is returned.
- Line 4 is not reached.

```
1 def do_something(n):
2     if n < 0:
3         return 0
4     return n ** 2
5
6 print(do_something(-4))
```

Lecture 3.2

Objects

Introduction

- So far the types of data we have worked with have been defined for us.
  - i.e. **floats**, **integers**, **strings**, etc.

- However, we might want to **define** our **own types**.
  - e.g. a "**student**" **type** that holds a **name** and **ID**.

- In this lecture we will learn about **objects** and how to **create** our **own** types using **classes**.

Lecture Overview

1. Objects
2. Creating Classes
3. References and Mutability

Objects

Objects Everywhere!

- An object consists of:
  - Related **pieces** of **data** (its value), and
  - **Functions** which use or **manipulate** that **data** (its methods).

- That is, an object *knows stuff* and can *do stuff*.

- In Python **almost everything** is an object.
  - Integers, strings, floats, and booleans are all objects.
  - If a variable can hold it, it's an object!

Date Objects

- We are going to use dates as an example to explore objects further.
- In Python, dates are part of the `datetime` module.
- The following line of code enables the creation of new date objects:
    - `from datetime import date`
- We'll learn more about importing modules in a future lecture.

Creating a Date Object

- A date has **three** key **pieces** of data: the **year**, the **month**, and the **day**.
    - It's a bit like **three variables** in **one**.

- The **action** of **creating** a new **object** is called instantiation.

- You could picture the date object representing 21/7/2012 like this:

year	2012
month	7
day	21

Creating a Date Object

- The code for instantiating a new object is similar to a **function call**.
- The **instantiated** object is returned.
- For a date object, the **arguments** are the **year**, **month**, and **day** (respectively).

```
>>> from datetime import date
>>> my_date = date(2012, 7, 21)
```

- Here we are instantiating an **object** to represent the date 21/7/2012.

Creating a Date Object

- If we ask Python about the type of **my_date**, we'll see something interesting.

```
>>> type(my_date)
<class 'datetime.date'>
```

- **my_date** is **not** an **int**, **str**, or **float**---it's something **else** entirely!

- We say that **my_date** is an **instance** of the `date` **class**.

Accessing Object Data

- We can access the **three** pieces of **data** stored in **my_date** by **name**.

- We say that **year**, **month**, and **day** are attributes of **my_date**.

- **Different** types of **objects** have **different attributes**.

```
>>> my_date.year
2012
>>> my_date.month
7
>>> my_date.day
21
```

Calling Methods

- A method is a **function** attached to an object that has **access** to the object's **value (data)**.

- The methods on **my_date** are **accessed** in a **similar** way to **attributes**.

```
>>> my_date.isoweekday()
6
```

- The `isoweekday` method **determines** which **day** of the week the **date falls** on.
  - Apparently 21/7/2012 was the **sixth** day of the week (Saturday).

What's The Point?

- **Grouping** related **data** (value) and behaviour (**methods**) makes code **more manageable**.
- Instead of keeping track of **multiple** different variables (year, month, day), we can **bundle** them **together**.
- We also have easy **access** to related **behaviour** (e.g. isoweekday).

Creating Classes

What is a Class?

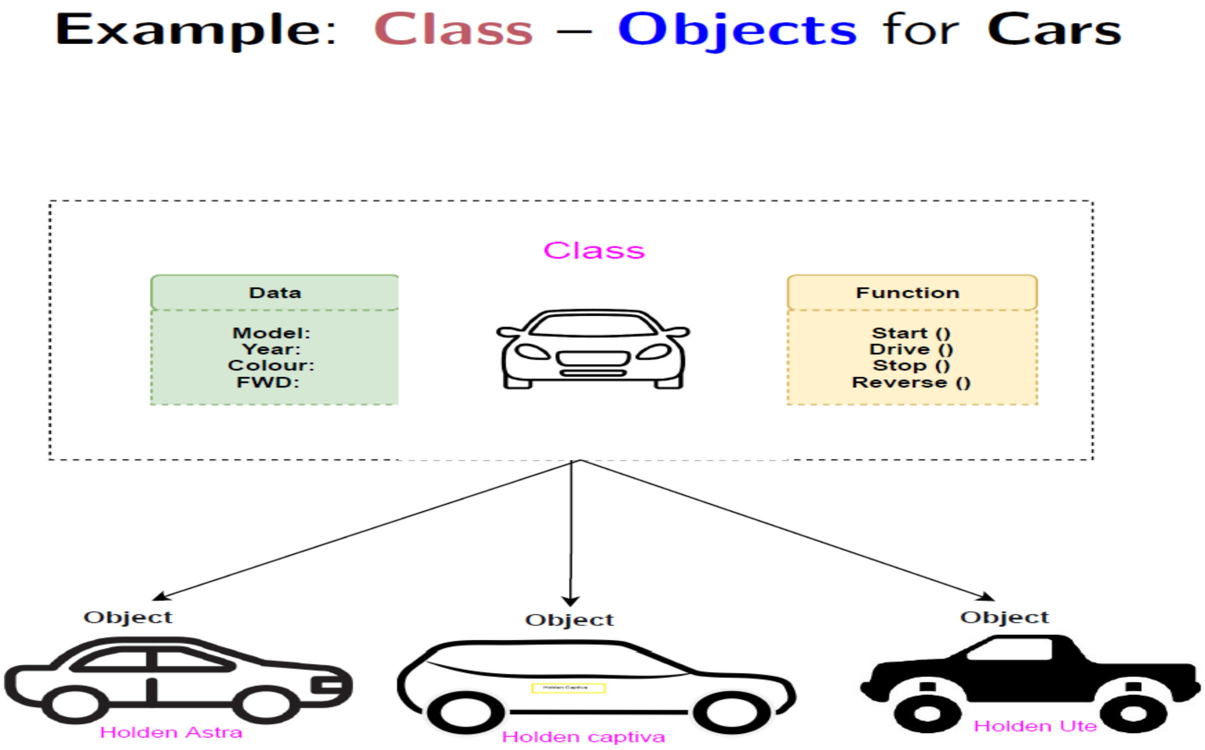
- A class is a **template** from which objects are instantiated (**created**).
  - A **date** class describes what a date is (i.e. that it is a **year**, a **month**, and a **day**).
  - A date object describes a particular date, like 21/7/2012.
- **One class** is typically used to create **many objects**.

Class Vs Object

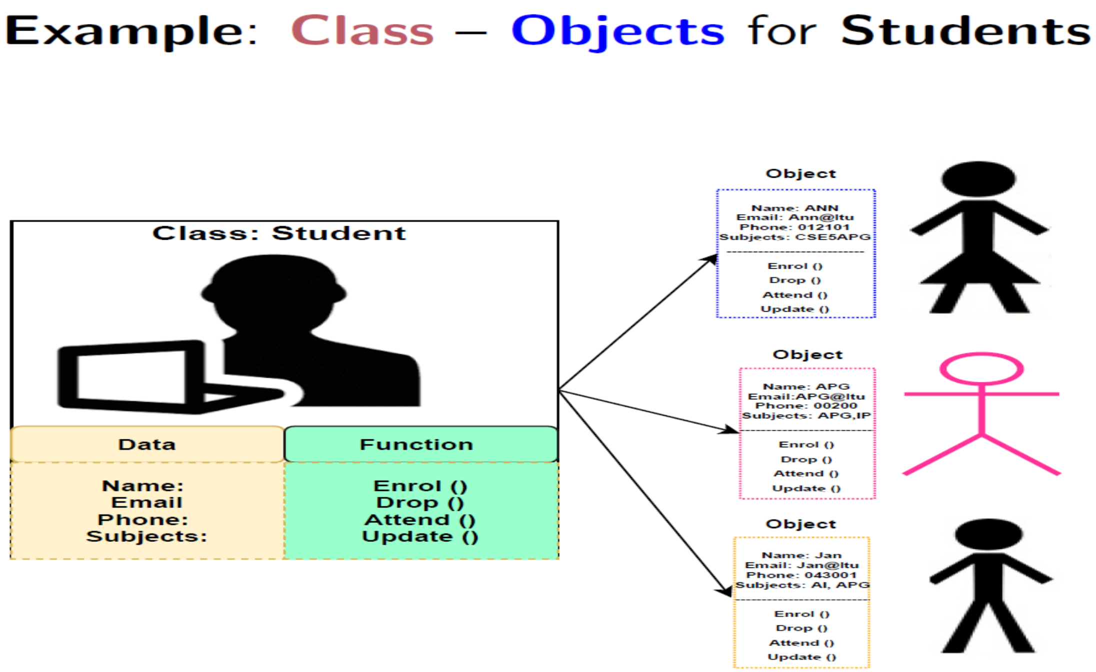
Classes vs Objects (Instances)

- Classes:
  - A class is used to create a **user-defined data structure**.
  - A class is **template** or a blueprint for making objects.
  - A class involves **data or variables** and **functions**.
- Objects:
  - An **object** (**instance**) is built from a class and contains real data. In other words, an object gets its data and functions from class.
  - We use this term to refer to, for example, people and organisations, to physical things like tables and chairs, and to non-tangible things like accounts, contracts, etc.

Example: Class - Object



Example: Class - Object

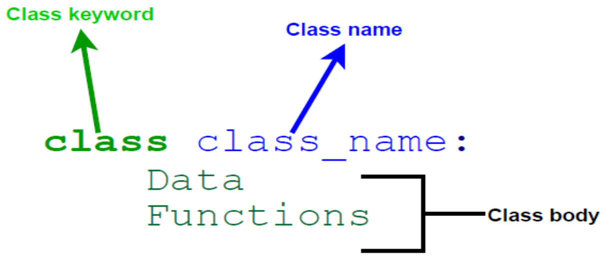


Creating a Custom Class

How to create a **Class**?

Example: Create a class

```
class class_name:
    Data
    Functions
```



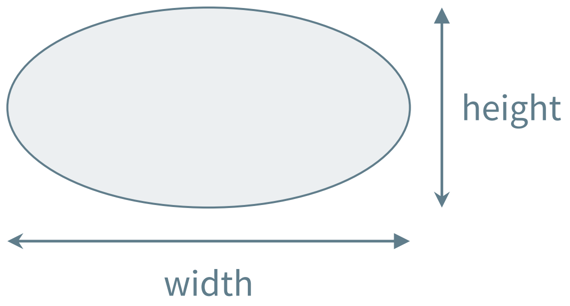
The **class** is defined with the **class** keyword followed by **class name** and ends with a **colon** (:). The **data** and **functions** are **indented** below the **class** keyword.

Python

```
>>> class_name
<class __console__.class_name>
```

Creating a Custom Class

- By creating a class, we have a custom template which we can use to instantiate objects.
- Let's create an Ellipse class that has a width and a height.



Creating an Ellipse Class

- __init__ is a special function called a constructor.
- The code in this **constructor** will be run every time an **Ellipse object** is **instantiated**.
- Behind the scenes, Python will provide the object instance as the **first parameter** (self).

```
class Ellipse:
    def __init__(self):
        self.width = 2

        self.height = 1
```

Creating an Ellipse Class

- When instantiating an Ellipse object, **two** variables called **width** and **height** will be attached to it.

- We call width and height instance variables.

- **width** and **height** will initially be **2** and **1**, respectively.

```
class Ellipse:
    def __init__(self):
        self.width = 2
        self.height = 1
```

Using Our Ellipse Class

```
>>> ell = Ellipse()
>>> type(ell)
<class '__main__.Ellipse'>
>>> ell.width
2
>>> ell.width = 3
>>> ell.width
3
```

- Notice that we can **assign** to instance **variables**, just like **regular** variables.

- Here we change the **width** of the ellipse from **2** to **3**.

Constructor Arguments

- Currently, if we want to **create** an **ellipse** with a particular **width** and **height**, we must:
  1. Instantiate an Ellipse. `ell = Ellipse()`
  2. Set the width. `ell.width = 5`
  3. Set the height. `ell.height = 7`

- To make things simpler, we can **rewrite** the **constructor** so that the **width** and **height** can be passed in as **arguments** when **instantiating** the object.

Constructor Arguments

```
class Ellipse:
    def __init__(self, w, h):
        self.width = w
        self.height = h
```

```
>>> ell = Ellipse(5, 7)
>>> ell.width
5
>>> ell.height
7
```

- Now **2 arguments** are expected when instantiating an Ellipse.
- These arguments are used to set the initial values of the **width** and **height** instance variables.

What is a Method?

- A method is a **function** which is **attached** to an object.

- Behind the scenes, Python itself will provide the object instance as the **first parameter** (self).
  - Sound familiar? This is because the **constructor**, __init__, is a **method**!

Creating a Method

```
class Ellipse:
    def __init__(self, w, h):
        self.width = w
        self.height = h
        self.PI=3.1415

    def calculate_area(self):
        return self.PI * self.width * self.height
```

- Here we have defined a calculate_area method which returns the area of the ellipse.

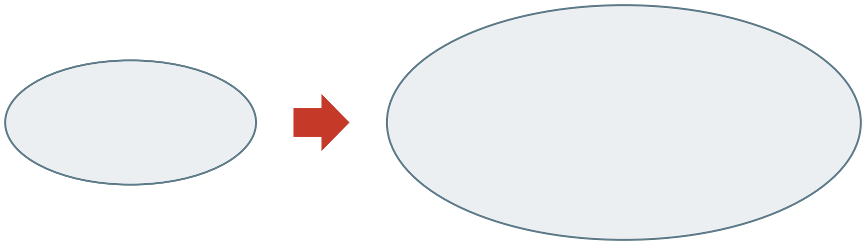
Using a Method

```
>>> ell = Ellipse(5, 7)
>>> ell.calculate_area()
109.95565
```

- Notice that we don't pass an argument for self, Python does that automatically.

Method Arguments

- A method can **accept** additional argument after self.
- A method can also **modify** instance variables.
- Let's add **another method** to the Ellipse class for **scaling** its **size**.



Method Arguments

```
class Ellipse:
    def __init__(self, w, h):
        self.width = w
        self.height = h

    def calculate_area(self):
        return 3.14159 * self.width * self.height

    def scale(self, s):
        self.width = self.width * s
        self.height = self.height * s
```

- The argument, **s**, determines the **scale** factor.

Method Arguments

```
>>> ell = Ellipse(3, 4)
>>> ell.calculate_area()
37.69907999999995
>>> ell.scale(2)
>>> ell.width
6
>>> ell.height
8
>>> ell.calculate_area()
150.79631999999998
```

- Notice that calling the scale method changes the **width** and **height** of our ellipse object.
  - This **changes** the value **returned** by **calculate_area**.

Check Your Understanding

Q. What are the instance variables of the Box class?

```
class Box:
    def __init__(self, width, height):
        self.area = width * height

        self.full = False
```

Check Your Understanding

Q. What are the instance variables of the Box class?

A. area and **full**.

- **width** and **height** are constructor **parameters**, **not** instance variables.
- The instance variables are **attached** to the object instance.

```
class Box:
    def __init__(self, width, height):
        self.area = width * height
        self.full = False
```

References and Mutability

Memory

- Every computer has memory.
    - RAM = random access memory.

- Memory provides a **temporary** storage area for programs to **store objects**.
    - When the program **finishes**, the objects in memory are **no longer accessible**.

References

- So far we have considered variables to be like **boxes** which hold **objects**, but this is **not** strictly **true**.
- In reality, the objects are **stored** in **memory** and variables **reference** those objects.
- A **variable** can **only reference** one **object**.
- The **same object** can be **referenced** by **many variables**.
- **Assigning** an object to a **variable causes** the **variable** to **reference** that object.
  - **It does not copy the object**.

Example: References

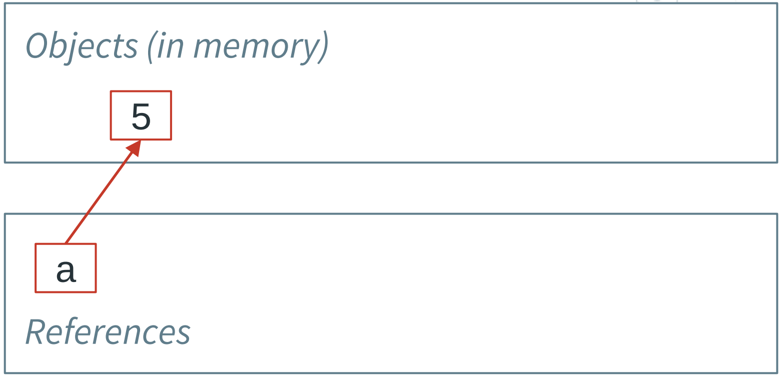
```
a = 5
b = a
c = 7
a = c
```

Objects (in memory)

References

Example: References

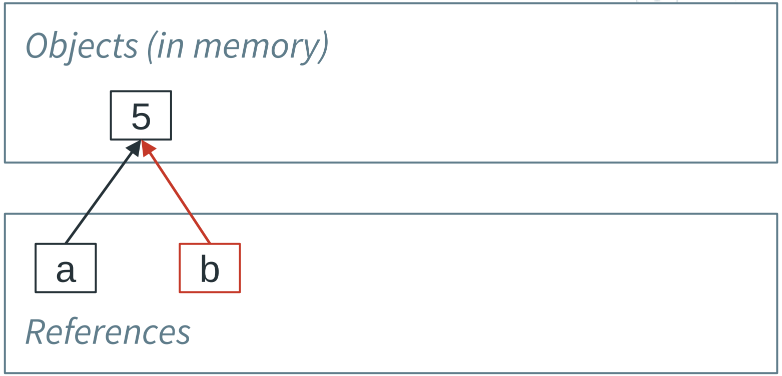
```
a = 5
b = a
c = 7
a = c
```



- The integer **5** is assigned to variable a.
  - a now references the integer **5** in memory.

Example: References

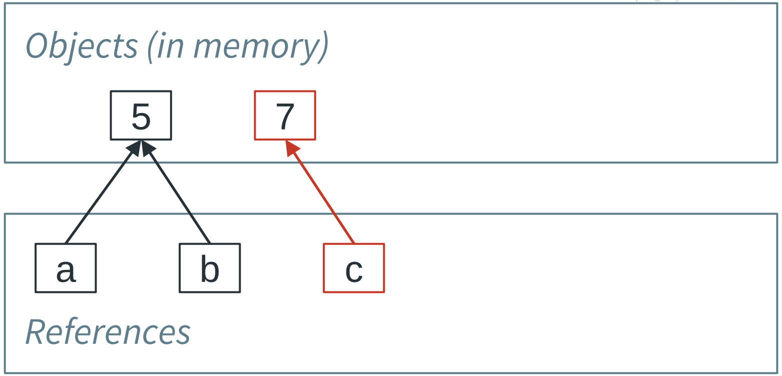
```
a = 5
b = a
c = 7
a = c
```



- The object referenced by **a** (which is the integer 5) is assigned to variable **b**.
  - Now both **b** and **a** reference the integer **5**.

Example: References

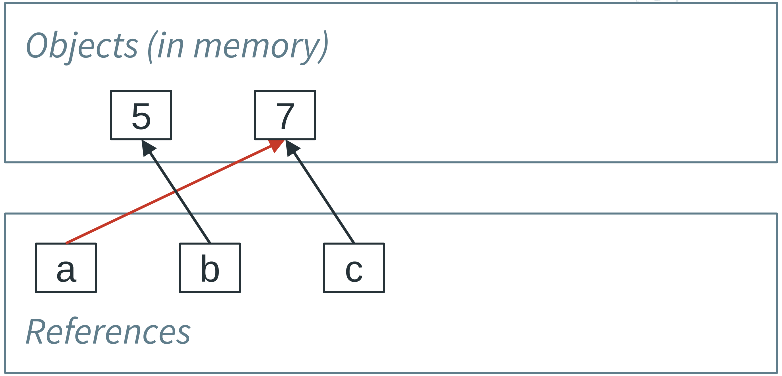
```
a = 5
b = a
c = 7
a = c
```



- The integer **7** is assigned to variable **c**.
  - **c** now references the integer **7** in memory.

Example: References

```
a = 5
b = a
c = 7
a = c
```



- The integer **7** is assigned to variable **c**.
  - **c** now references the integer **7** in memory.

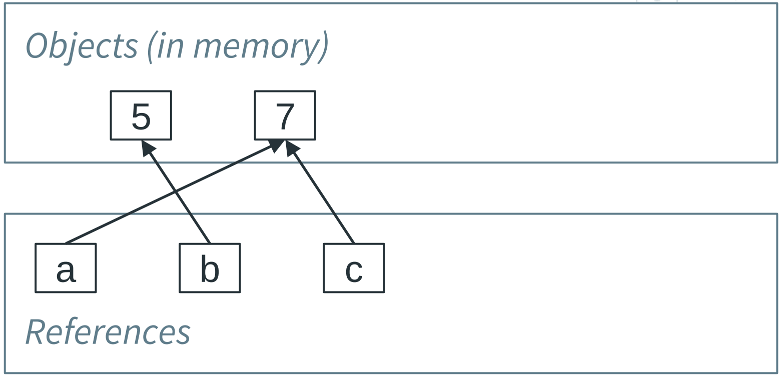
Objects Without References

- An **object** can only be **accessed** if there is a **reference** to it.
  - That is, when there are **no references** to an object, that object is **no longer** accessible by the program.

- Python will **automatically remove** objects **without** references from memory.
  - This helps to **free up space** in memory.

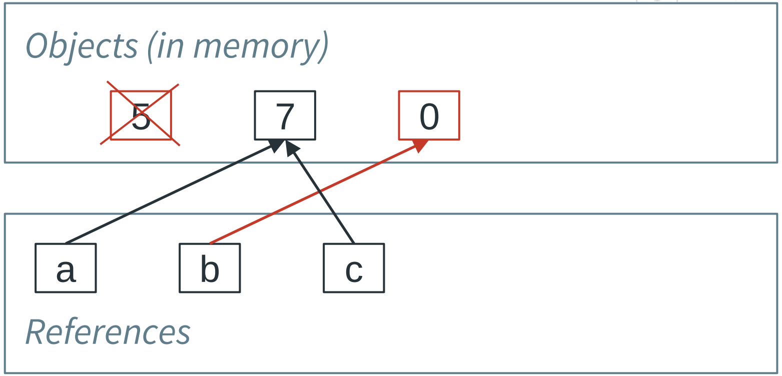
Example: Objects Without References

```
b = 0
```



Example: Objects Without References

```
b = 0
```



- **b** now references the integer **0** in memory.
- There are **no** references left to the integer **5**.
    - That object can be automatically **removed** by Python.

Summary of References

- **Objects** are **stored** in **memory**.
- **Variables** reference **objects**.
  - A **variable** can only reference **one object**.
  - The **same** object can be **referenced** by **many** variables.
- **Assignment** can be used to **change** which object a variable references.
- Objects with **no** references are **inaccessible**.

Immutable Objects

- An immutable object **cannot** have its value changed.
- Before this lecture, all of the types we've dealt with have been **mutable** (int, float, etc.).
- When using **immutable** objects in a computation, a new object is created to represent the result.
  - The original objects do not change.

Example: Immutable Objects

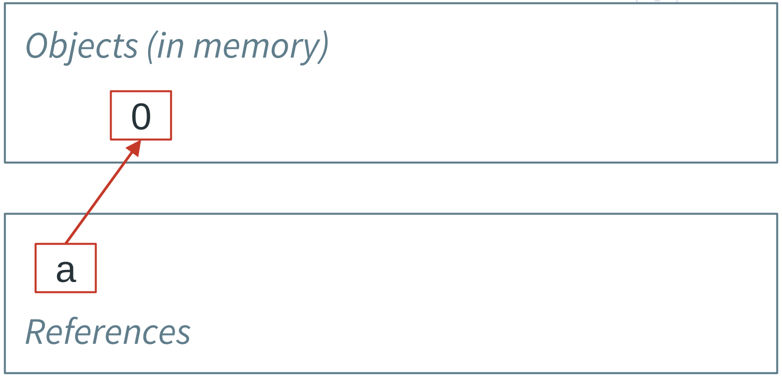
```python
a = 0
b = a
b = b + 1
```

Objects (in memory)

References

Example: Immutable Objects

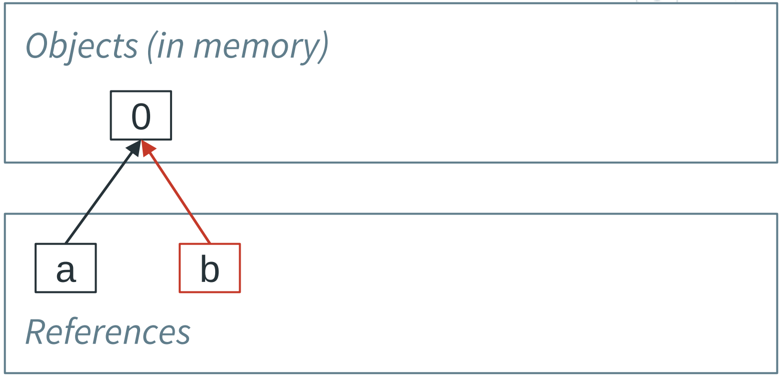
```
a = 0
b = a
b = b + 1
```



- The integer 0 is assigned to variable a.
    - a now references the integer 0 in memory.

Example: Immutable Objects

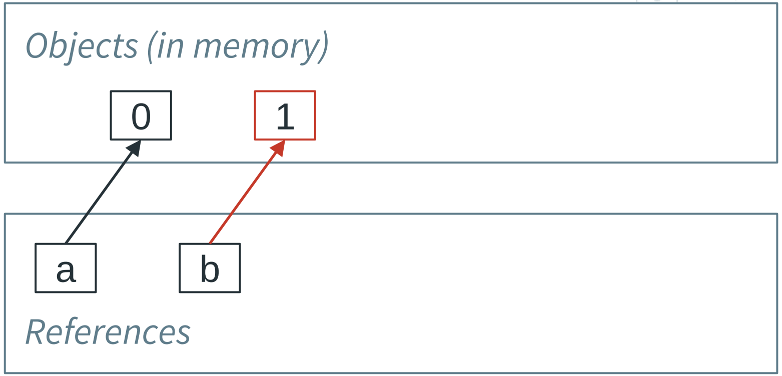
```
a = 0
b = a
b = b + 1
```



- The object reference by a (the integer 0) is assigned to variable b.
  - Now both b and a reference the integer 0.

Example: Immutable Objects

```
a = 0
b = a
b = b + 1
```



- The new integer 1 (result of b + 1) is assigned to variable b.
  - b now references the integer **1** in memory.
  - a continues to reference the integer **0**.
    - i.e. the original object still has its original value, **0**.

Mutable Objects

- Mutable objects ***can* be changed**.
  - We saw this earlier when we changed the **width** and **height** of our **Ellipse** object.
- Object **instances** created from custom classes are **mutable**.
- If **multiple** variables refer to the same **object**, **mutating** the object will **affect all of them**.

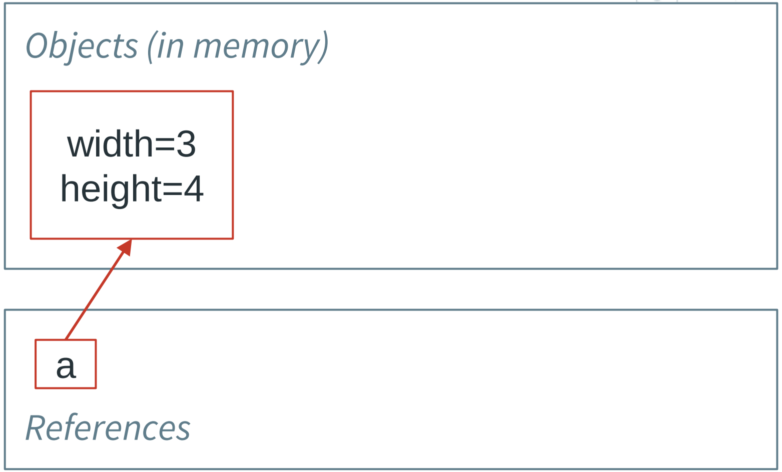
Example: Mutable Objects

```
>>> a = Ellipse(3, 4)
>>> b = a
>>> a.scale(2)
>>> a.width
6
>>> b.width
6
```

- **a** and **b** reference the **same** Ellipse **object**.
- As long as they hold the same **reference**, **a** and **b** can be used **interchangeably**.

Example: Mutable Objects

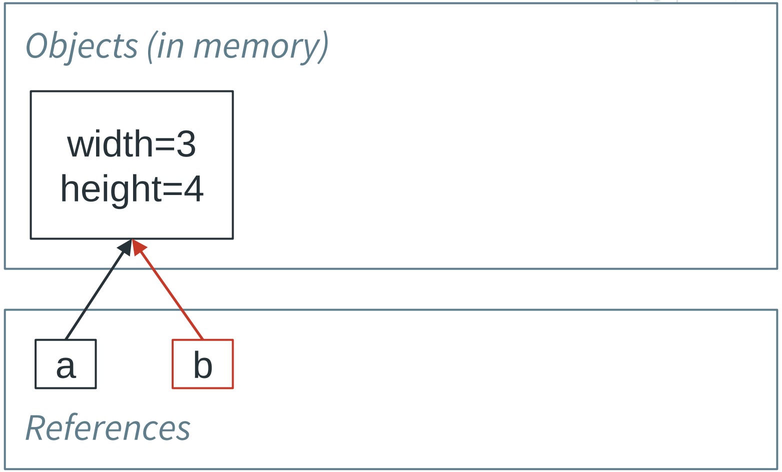
```python
a = Ellipse(3, 4)
b = a
a.scale(2)
```



- The Ellipse object is assigned to variable a.
    - a now references the ellipse object in memory.

Example: Mutable Objects

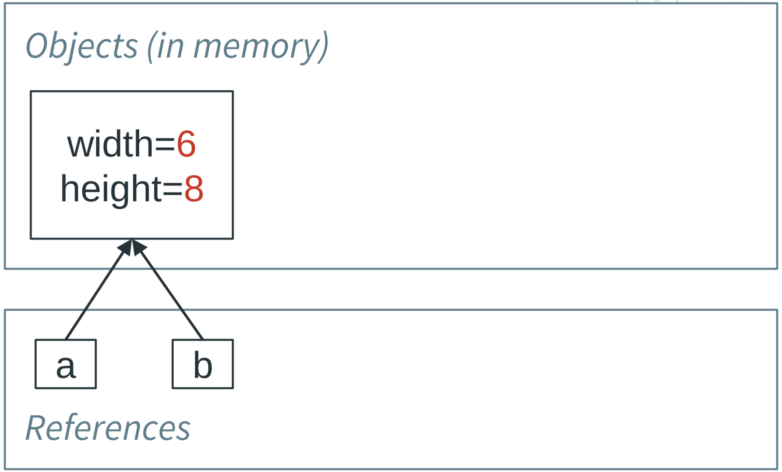
```
a = Ellipse(3, 4)
b = a
a.scale(2)
```



- The object referenced by b is assigned to variable a.
  - Now both b and a reference the same Ellipse object.

Example: Mutable Objects

```
a = Ellipse(3, 4)
b = a
a.scale(2)
```



- The scale method mutates the Ellipse object.
- The effect will be seen by both a and b.

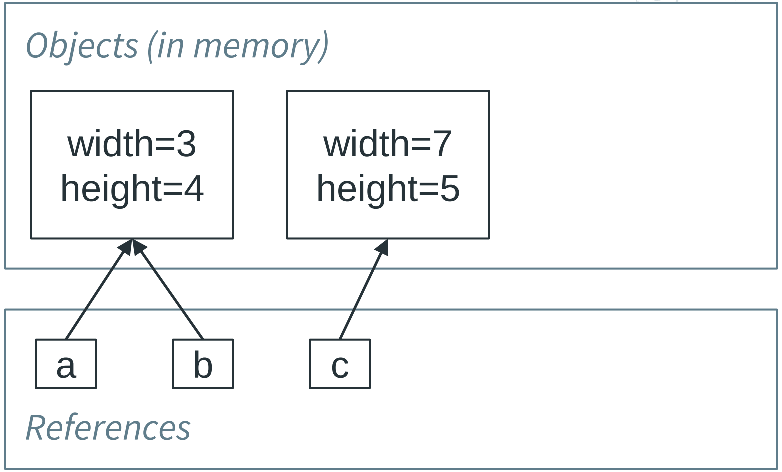
Comparing References

- In Python, the is **keyword** can be used to check whether **two** variables **reference** the **same** object.

- This is equivalent to checking whether two **arrows point** to the **same** object in our **diagrams**.

Example: Comparing References

```
>>> a = Ellipse(3, 4)
>>> b = a
>>> c = Ellipse(7, 5)
>>> a is b
True
>>> a is c
False
```



Multiple Instances

- Using a class to instantiate **multiple** objects will result in **different** object **instances**.

- They are **different** objects even if they have **the same value**.
  - That is, they will **occupy** different locations in memory.
  - **Mutating** one will **not** affect the other (in general).

Example: Multiple Instances

```
>>> a = Ellipse(3, 4)
>>> b = Ellipse(3, 4)
>>> a.scale(2)
>>> a.width
6
>>> b.width
3
>>> a is b
False
```

- **a** and **b** reference **different** Ellipse **objects**.
- **Mutating a** does not affect **b**.

Example: Multiple Instances

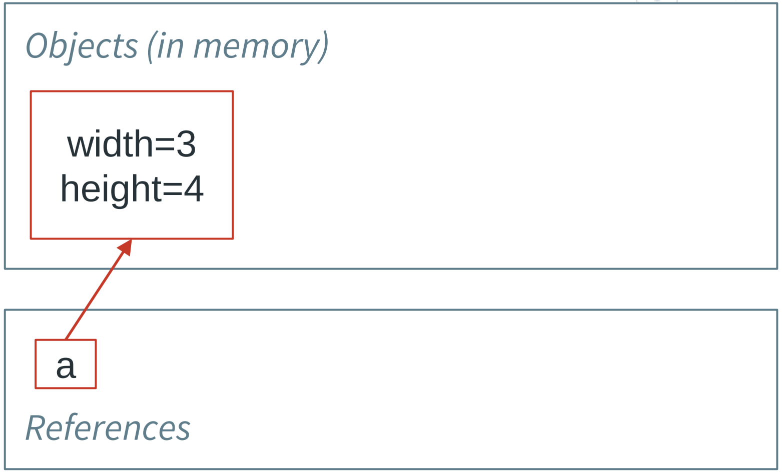
```
a = Ellipse(3, 4)
b = Ellipse(3, 4)
a.scale(2)
```

Objects (in memory)

References

Example: Multiple Instances

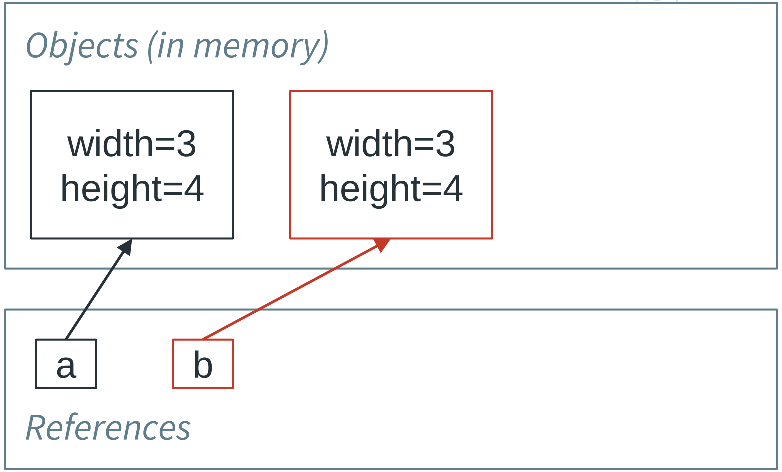
```
a = Ellipse(3, 4)
b = Ellipse(3, 4)
a.scale(2)
```



- The Ellipse object is assigned to variable a.
  - a now references the ellipse object in memory.

Example: Multiple Instances

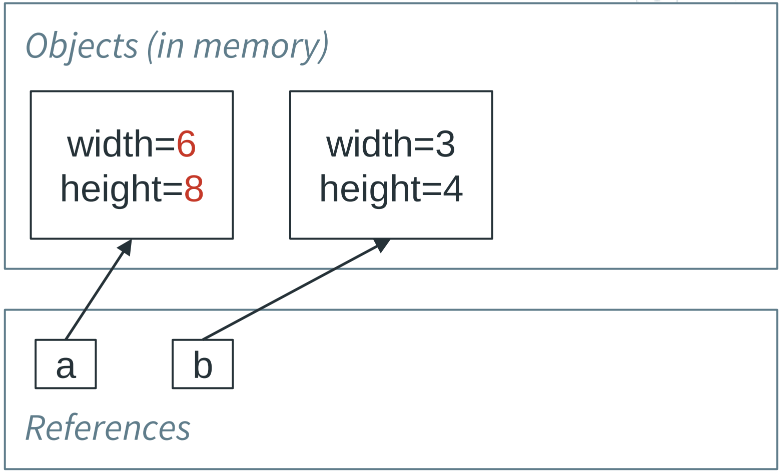
```
a = Ellipse(3, 4)
b = Ellipse(3, 4)
a.scale(2)
```



- A *different* Ellipse object is instantiated and assigned to variable b.
  - b now references the second Ellipse object in memory.

Example: Multiple Instances

```
a = Ellipse(3, 4)
b = Ellipse(3, 4)
a.scale(2)
```



- The scale method mutates the first Ellipse object.
- The effect will be seen by a only.

Check Your Understanding

Q. Assuming that the Ellipse class has been defined, what will be the outputs of the shown program?

```python
var1 = Ellipse(2, 2)
var2 = Ellipse(3, 4)
var1 = var2
var1.height = 5
print(var1.width)
print(var1.height)
```

Check Your Understanding

Q. Assuming that the Ellipse class has been defined, what will be the outputs of the shown program?

A. 3 and 5.

```
var1 = Ellipse(2, 2)
var2 = Ellipse(3, 4)
var1 = var2
var1.height = 5
print(var1.width)
print(var1.height)
```

Check Your Understanding

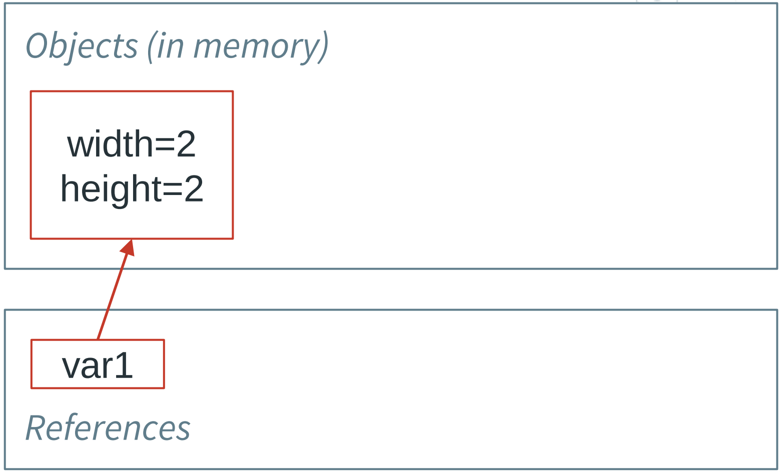
```
var1 = Ellipse(2, 2)
var2 = Ellipse(3, 4)
var1 = var2
var1.height = 5
```

Objects (in memory)

References

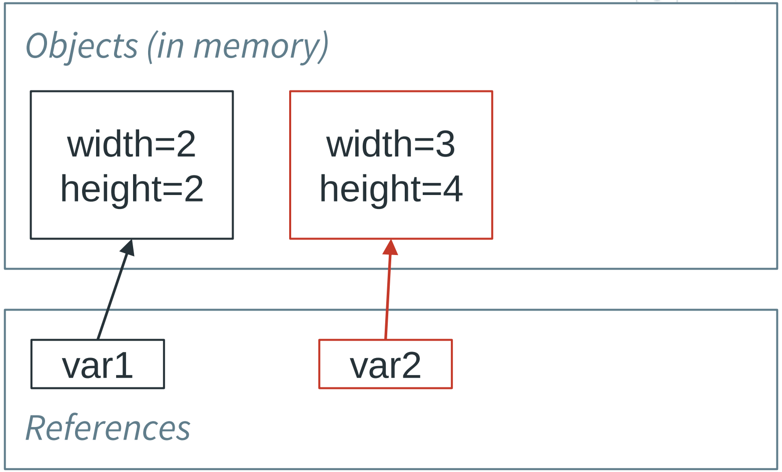
Check Your Understanding

```
var1 = Ellipse(2, 2)
var2 = Ellipse(3, 4)
var1 = var2
var1.height = 5
```



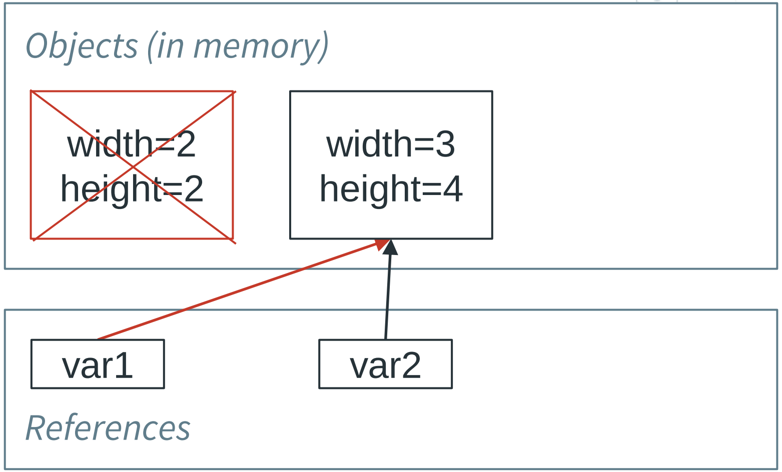
Check Your Understanding

```
var1 = Ellipse(2, 2)
var2 = Ellipse(3, 4)
var1 = var2
var1.height = 5
```



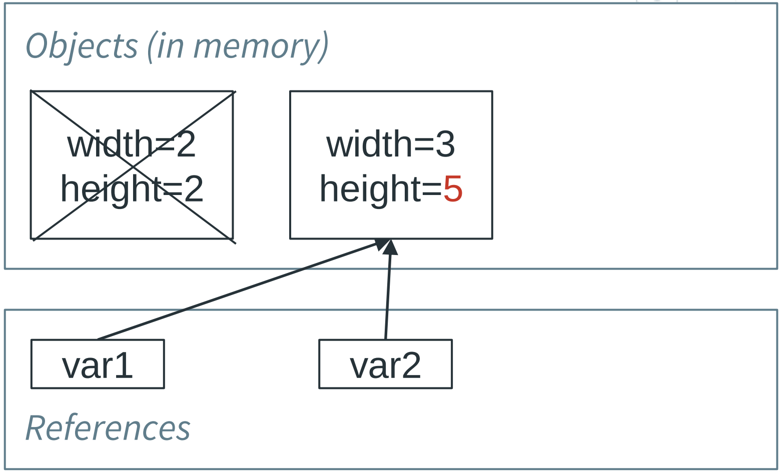
Check Your Understanding

```
var1 = Ellipse(2, 2)
var2 = Ellipse(3, 4)
var1 = var2
var1.height = 5
```



Check Your Understanding

```
var1 = Ellipse(2, 2)
var2 = Ellipse(3, 4)
var1 = var2
var1.height = 5
```



- So var1.width is 3 and var1.height is 5.

Example

Example: Write a Python code to create bank account using class data structure

```
class BankAccount:
    def __init__(self, accountNr, customerName, balance = 0):
        self.accountNr = accountNr
        self.customerName = customerName
        self.balance = balance
# main
# Create an account
# Define an object name as a1
a1 = BankAccount("A10", "Bob", 1000)

# Get the account number
print("a1.accountNr:", a1.accountNr)
>>> "A10"

# Get the customer name
print("a1.customerName:", a1.customerName)
>>> "Bob"

# Get the balance
print("a1.balance:", a1.balance)
>>> 1000
```

Lecture 3.3

Strings

Topic 3.3 and 3.4 Intended Learning Outcomes

- By the end of the week you should be able to:
  - Express **text** in Python using different kinds of string literals and escape sequences,
  - Build strings in a neater and more flexible way using f-strings,
  - **Read** and **write** data from text files, and
  - Perform various file system operations.

Lecture Overview

1. More String Literals
2. Manipulating Strings
3. f-strings and Formatting

More String Literals

What The Literal...

- Up until this point, we've been using **single** quotes around text to create string literals.
  - e.g. `'Programming’`
- But if a single quote indicates the **end** of the string, how do we **include** a **single** quote ***inside*** the string?
  - `'This doesn't work’`
- And what about other special characters, like **newlines** and **tabs**?

Escape Character

- In Python the **backslash**, \, is an escape character which changes how the character after it is interpreted **inside** a string.

- Placing a backslash **before** a **single** quote tells Python *"this isn't the end of the string, it's just a **single** quotation mark"*.

```
>>> print('Quotes aren\'t a problem')
Quotes aren't a problem
```

- `\'` is an example of an escape sequence.

Common Escape Sequences

OPERATOR	NAME
`\\`	One backslash (\\)
`\'`	Single quote (')
`\"`	Double quote (")
`\n`	Newline
`\t`	Horizontal tab

Example: Escape Sequences

```
>>> print('First line\nSecond line')
First line
Second line

>>> print('\\\'o\'/')
\'o’/

>>> print('"Free"')
"Free“

>>> print('\"Free\"')

"Free"
```

Alternative String Literal Syntax

- Although you can use **escape sequences** to express any **string** you can think of, it can start to look a bit **ugly**.
  - e.g. `'\'\'\'\n\'\'\‘’`

- Python provides **alternative** ways of expressing string literals which you can **choose between**.

Double Quote Strings

- You can use **double** quotes instead of **single** quotes.
  - Changes which type of quotes need to be escaped.

- Useful when you need to include apostrophes.
  - e.g. `"I don't need to escape"`

- The **trade-off** is that **double** quotes now need to be escaped.
  - e.g. `"It isn't the \"real\" you"`

Multiline Strings

- You can write a string that spans **multiple** lines by using **three** single/double quotes.

- This avoids the need to use **\n** to separate lines.
  - You can still use **\n** inside multiline strings if you want, though.

Example: Multiline Strings

```
>>> print("""One
... Two""")
One
Two
>>> print('''"1"\n"2"
... "3"''')
"1"
"2"
"3"
```

Indentation with Multiline Strings

- Note that the **space** included in the string **matters**, so things can look a bit **weird** when code is indented.

```
ring = True
if ring:
    print('''ding
    dong''')
```

Output:

```
ding
    dong
```

Indentation with Multiline Strings

- Python allows you to **break** the usual **indentation** rules inside **multiline strings** to work around this problem.

```
ring = True
if ring:
    print('''ding
dong''')
```

Output:

```
ding
dong
```

Check Your Understanding

Q. Use a double-quoted, single-line string literal to express the following text:

```
Is it cheap,
Or "cheap"?
```

Check Your Understanding

Q. Use a double-quoted, single-line string literal to express the following text:

```
Is it cheap,
Or "cheap"?
```

A.

```
"Is it cheap,\nOr \"cheap\"?"
```

- **\n** represents the newline.
- \" represents double quotes.

Manipulating Strings

Character Sequences

- A string is a sequence of characters.
- The length of a string is the number of characters in the string.
  - You can use the len built-in function to get this value.
  - e.g. `len('Avocado')` returns 7.
- An escape sequence counts as 1 character.
  - e.g. `len('A\nB')` returns 3.

Character Sequences

- The index of a character **indicates** its **position** in the **string**.
- **Indices start** at **0** (so the **first** character is at **index** 0).
- The last index is the length of the string minus 1.

Character Sequences

'Star Wars'

S	t	a	r		W	a	r	s
0	1	2	3	4	5	6	7	8

- The character at **index** **1** is `t`.
- The character at index **4** is the **space** character, ` `.

Negative Indices

`'Star Wars'`

S	t	a	r		W	a	r	s
-9	-8	-7	-6	-5	-4	-3	-2	-1

- In Python, you can use **negative** indices to **count** backwards from the **end** of the string.

- The character at **index -1** is the last character of the string (in this case, `s`).

Indexing a String

- You can use square brackets [] to retrieve a character by its index.
- This is called indexing a string.
- You can use **normal** or **negative** indices.

```
>>> movie = 'Star Wars'
>>> movie[0]
'S'
>>> movie[-2]
'r'
```

Slicing a String

- You can also use **square brackets** to retrieve a segment of a string (a substring).

- This is called **slicing** a string.

```
>>> movie = 'Star Wars'
>>> movie[0:3]
'Sta'
>>> movie[-4:-1]
'War'
```

- Notice that the character at the **first** index is ***included*** but the character at the **last** index is ***excluded***.

Slicing a String

- If you **omit** the **first** index, **all** characters from the **start** of the string are **returned**.

- If you **omit** the **second** index while slicing, **all** characters to the **end** of the string are **returned**.

```
>>> movie = 'Star Wars'
>>> movie[:4]
'Star'
>>> movie[-4:]
'Wars'
```

Looping Over Characters

- Since a string is a **sequence** of characters, it can be used in a for **loop** to **iterate** over each character.
- Each item is a string of **length 1** containing a single **character**, **starting** with the first character.

Example: Looping Over Characters

```
dna = 'GATTACA'
for nucleotide in dna:
    if nucleotide == 'G' or nucleotide == 'A':
        print(nucleotide)
```

Output:

```
G
A
A
A
```

Searching

- The `find` string method returns the **position** of the **first occurrence** of a smaller substring within a larger string.

- The position returned is the **index** of the **first** character of the **matching** part of the string.

```
>>> s = "Where's Wally?"
>>> s.find('W')
0
>>> s.find('Wally')
8
```

Failing to Find

- If the string does **not** contain the substring, **-1 is returned**.

```
>>> s = "Where's Wally?"
>>> s.find('Odlaw')
-1
>>> s.find('w')
-1
```

Searching Backwards

- You can find the **last** instance of the substring (**instead** of the first) but using the `rfind` string method.

```
>>> s = "Where's Wally?"
>>> s.find('e')
2
>>> s.rfind('e')
4
```

Case and String Comparison

- It is important to keep in mind that the **case** of letters **matters** when **comparing** strings.

- An **uppercase** letter (e.g. 'A') is **different** to a **lowercase** letter (e.g. 'a').

```
>>> 'earth' == 'Earth'
False
>>> 'grand canyon'.find('canyon')
6
>>> 'Grand Canyon'.find('canyon')
-1
```

Case and String Comparison

- To ignore case, you can **convert** all of the letters to either **lowercase** or **uppercase** **before** **comparing**.
  - `lower` **converts** all **letters** to **lowercase**.
  - `upper` **converts** all **letters** to **UPPERCASE**.

```
>>> 'earTH'.lower() == 'Earth'.lower()
True
>>> 'grand canyon'.upper().find('CANYON')
6
>>> 'grand canyon'.upper().find('canyon')
-1
```

String Replacement

- The `replace` method can be used to **replace parts** of a string. It **takes** 2 arguments:
  - **First**: The **substring** to **replace**.
  - **Second**: The **replacement**.

```
>>> s = "Where's Wally?"
>>> s.replace('ly', 'do')
"Where's Waldo?“

>>> s.replace('W', 'R')
"Rhere's Rally?“

>>> s

"Where's Wally?"
```

- A new string is returned (the original string is **not** modified).

String Replacement

- If the substring **isn't** found, the string is returned **unchanged**.

- The **second** argument (the replacement) can be an **empty** string.
  - Using an **empty** string will result in the **found** substrings being **replaced** with **nothing** (i.e. removed).

Example: Removing Vowels

```python
# File: remove_vowels.py
vowels = 'aeiouAEIOU'
s = input('Enter some text: ')
for vowel in vowels:
    s = s.replace(vowel, '')
print(s)
```

Example run:

```
$ python remove_vowels.py
Enter some text: A fox is orange.
 fx s rng.
```

Check Your Understanding

Q. What is the result of the following Python expression?

```
'No worries, Oliver'.rfind('o')
```

Check Your Understanding

Q. What is the result of the following Python expression?

```
'No worries, Oliver'.rfind('o')
```

A. 4.

- There are two instances of the substring 'o'.
  - Capital 'O' is different to lowercase 'o’.
- **rfind** searches backwards, so it finds the last 'o’.
- The index of the last 'o' is 4.

f-strings and Formatting

String Concatenation Can Be Ugly (and it's not even beautiful on the inside)

- So far we have built strings using the + operator.
- We used **str()** to convert other types of data into strings prior to concatenation.
- The code from this can become messy.

```
acc = 1
bal = 23.0
print('Account ' + str(acc) + ' balance is $' + str(bal))
```

f-strings

- Python provides f-strings as a convenient way of building strings.

- An **f-string** is a string literal **prefixed** with an `f`.
- When writing an **f-string**, you can include Python expressions by using curly brackets.
  - This part of an f-string is called a replacement field.
- **f-strings** reduce the need for + and str().

f-strings

```python
# With string concatenation
acc = 1
bal = 23.0
print('Account ' + str(acc) + ' balance is $' + str(bal))

# With an f-string
acc = 1
bal = 23.0
print(f'Account {acc} balance is ${bal}')
```

- Note that the f-string has an `f` before the first quote.
- `{acc}` and `{bal}` are replacement fields.

Formatting Values

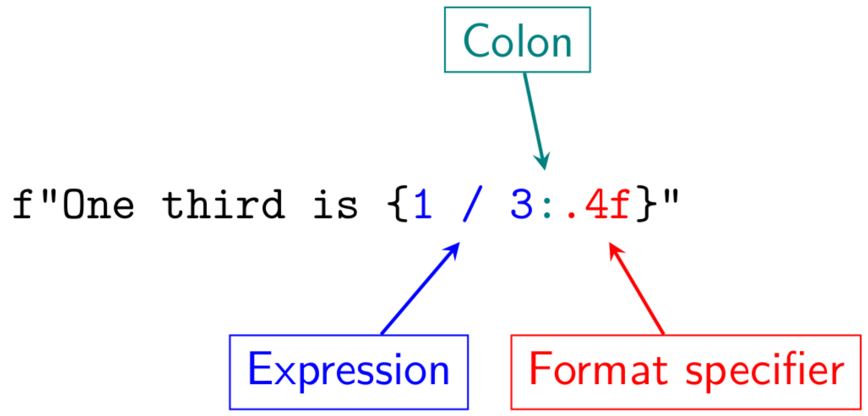
- There are multiple ways of expressing a number.
    - e.g. the number **5** can be written as 5, 5.0, 5.00, etc.

- A programmer should be able to choose how a number is displayed **depending** on the **application**.
    - For example, dollar amounts are usually expressed using two decimal places (e.g. $1.99).
    - Precise distance measurements might be expressed using **many decimal places**.

Format Specifiers

- You can specify how values are to be formatted in an **f-string**.
- This is achieved by including a format specifier in a replacement field.
- Format specifiers are actually quite in-depth and allow for lots of customisation.
- We will now cover some common kinds of format specifiers.

Format Specifiers



- A format specifier can be added to a replacement field.
- The format specifier is separated from the expression using a **colon**, `:`.

Number of Decimal Places

- You can specify the **number of decimal** places shown.
- Appropriate **rounding** will be applied.
- Trailing **zeros** will be added if necessary.

```
>>> x = 0.6666
>>> f'{x:.3f}'
'0.667'
>>> f'{x:.0f}'
'1'
>>> f'{x:.6f}'
'0.666600'
```

Number of Decimal Places

- The letter "f" in the format specifier indicates that the value should be presented as a number with a fixed-length fractional part.
- This is called the presentation type or format code.

```
>>> x = 0.6666
>>> f'{x:.3f}'
'0.667'
>>> f'{x:.0f}'
'1'
>>> f'{x:.6f}'
'0.666600'
```

Number of Decimal Places

- The ".#" in the format **specifier** indicates the **number** of decimal places.

```
>>> x = 0.6666
>>> f'{x:.3f}'
'0.667'
>>> f'{x:.0f}'
'1'
>>> f'{x:.6f}'
'0.666600'
```

Width

- The minimum **space** that the **formatted** value will **occupy** can be **specified**.
    - The "**width**" in terms of number of characters.
- The extra room is filled with **space** characters.
- Useful for aligning program output.

```
>>> f'{123:6d}'
'   123'
>>> f'{123:2d}'
'123'
>>> f'{0.11:6.2f}'
'  0.11'
```

Width

- Notice that **spaces** are **added** to the start to ensure that the minimum width is met.
- Strings that are too long are not affected.

```
>>> f'{123:6d}'
'   123'
>>> f'{123:2d}'
'123'
>>> f'{0.11:6.2f}'
'  0.11'
```

Width

- Using the "**d**" presentation type indicates that the value should be presented as a whole number **without** a **fractional** part.

```
>>> f'{123:6d}'
'   123'
>>> f'{123:2d}'
'123'
>>> f'{0.11:6.2f}'
'  0.11'
```

- You can only use "d" when the value is an integer.

```
>>> f'{12.3:6d}'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unknown format code 'd' for object of type 'float'
```

Width

- Here’s a times table program that produces nicely **aligned output** using format specifiers.

- Can you see how the spaces **automatically added** to **smaller** numbers result in visually pleasing **formatting**?

```python
a = int(input('Times table: '))
for b in range(1, 13):
    print(f'{a:2d} x {b:2d} = {a * b:3d}')
```

```
Times table: 11
11 x  1 =  11
11 x  2 =  22
11 x  3 =  33
11 x  4 =  44
11 x  5 =  55
11 x  6 =  66
11 x  7 =  77
11 x  8 =  88
11 x  9 =  99
11 x 10 = 110
11 x 11 = 121
11 x 12 = 132
```

Width for String Values

- String values can also be formatted with a **minimum** width.

- In order to format a string, you should use the "**s**" **presentation** type instead of "**d**" or "**f**".

- You can **only** use "**s**" when the **value** is a string.

```
>>> f'{"hello":8s}'
'hello   '

>>> f'{96:8s}'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unknown format code 's' for object of type 'int'
```

Alignment

- Notice that, unlike numbers, strings are left-aligned by default rather than right-aligned.
  - Spaces are added to the end, not the start.

- You can force a particular alignment using < for left alignment and > for right alignment.

```
>>> f'{"hello":8s}'
'hello   '
>>> f'{"hello":>8s}'
'   hello'
>>> f'{42:6d}'
'    42'
>>> f'{42:<6d}'
'42    '
```

Check Your Understanding

Q. What will the output of the shown program be?

```
speed_kph = 40.0
time_h = 0.5
print(f'|{time_h * speed_kph:<6.2f}|')
```

Check Your Understanding

Q. What will the output of the shown program be?

```
speed_kph = 40.0
time_h = 0.5
print(f'|{time_h * speed_kph:<6.2f}|')
```

A. `|20.00 |`

(note the space between the 0 and the |).

Check Your Understanding

Q. What will the output of the shown program be?

```
speed_kph = 40.0
time_h = 0.5
print(f'|{time_h * speed_kph:<6.2f}|')
```

A. `|20.00 |`

(note the space between the 0 and the |).

- The result of the expression is 20.0.

Check Your Understanding

Q. What will the output of the shown program be?

```
speed_kph = 40.0
time_h = 0.5
print(f'|{time_h * speed_kph:<6.2f}|')
```

A. `|20.00 |`

(note the space between the 0 and the |).

- The .2f part of the format specifier specifies presentation with 2 decimal places (i.e. 20.00).

Check Your Understanding

Q. What will the output of the shown program be?

```
speed_kph = 40.0
time_h = 0.5
print(f'|{time_h * speed_kph:<6.2f}|')
```

A. `|20.00 |`

(note the space between the 0 and the |).

- The 6 part of the format specifier specifies that anything less than 6 characters wide will have spaces added to make it 6 characters wide.

Check Your Understanding

Q. What will the output of the shown program be?

```
speed_kph = 40.0
time_h = 0.5
print(f'|{time_h * speed_kph:<6.2f}|')
```

A. `|20.00 |`

(note the space between the 0 and the |).

- The < part specifies that the spaces should be added to the end (left-aligned).
- Since 20.00 is 5 characters wide, the result of <6 is that one space character will be added to the end.

Check Your Understanding

Q. What will the output of the shown program be?

```
speed_kph = 40.0
time_h = 0.5
print(f'|{time_h * speed_kph:<6.2f}|')
```

A. `|20.00 |`

(note the space between the 0 and the |).

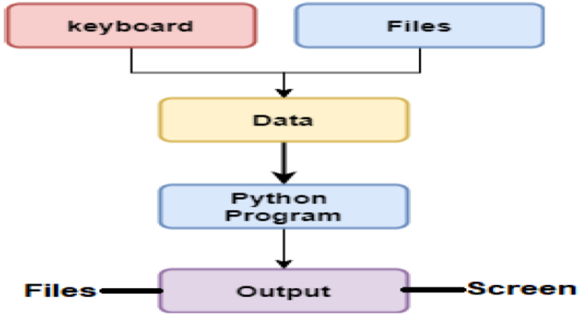
- The pipe characters (|) don't have any special meaning, they are just part of the string.

Lecture 3.4

Files

Introduction

So far we have learned that we can use the `input()` function to read data from the user (keyboard) and `print()` function to display the results on the screen. In Python, we can also read data from files or print data into files.



Files are named locations on PC hard disk to store different information such as data, programs, texts, and images.

```
Data = input ("Enter Data: ") # read from keyboard
print (Data) # display data at screen
```

```
Data = Read data from Files. # get data from a file
Write Data to Files         # save data in a file
```

Introduction

- Up until this point, the ways in which our programs handled input and output have been limited.
  - Input was always entered by the user.
  - Output was always displayed to the user.
- If the user want to **save the output**, they would need to copy it or write it down.
- This is quite frankly **lame**, and I wouldn't want to be friends with a program that made me copy **hundreds** of **lines** of **output** manually.

Introduction

- We also had no way of saving data for future runs of the program.
  - i.e. the program had no way of remembering anything about previous runs.

- In this lecture, we will learn about **reading** and **writing** text files as an alternative form of **input** and **output**.

- This will also allow us to write programs which process **existing** data **stored** in text files.

Lecture Overview

1. Reading Text Files
2. Writing Text Files
3. File System Operations

Reading Text Files

Text Files

- A text file is like a string saved on a computer's storage drive (hard drive or solid state drive).
- It's very likely that you have come across text files before.
- Some examples of files which are text files include .txt files, CSV files, HTML pages, and Python source code.
- As a general rule of thumb, if you can edit a file in Notepad, it’s a text file.

Binary Files

- Files which are not text files are called binary files.
- Binary files do not fundamentally represent their data using human-readable characters.
- Some examples of files which are binary files (and not text files) are JPEG images, MP3 songs, ZIP archives, and Word documents.

Opening Files

- In order to read from or write to a file, it must be opened.
  - Python's `open` built-in function opens a file.

- In Python, opening a file successfully will result in a file object which represents that file.

Opening Files

```
>>> f = open('colours.txt')
>>> f
<_io.TextIOWrapper name='colours.txt' mode='r' encoding='UTF-8'>
```

File Objects

- A file object has the information Python requires to access the contents of a file.
  - It also contains the name of the file.

- A file object **does not** represent the data contained within the file.
  - Data is accessed using methods on the file object for reading and writing data.

Failing to Open

- A call to `open` can fail, which will result in an error being raised.
    - e.g. if the file doesn't exist.
- For now, we will simply assume that the files we are opening are accessible.
- In a future lecture, we will discuss how to handle errors.

```
>>> open('404.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
'404.txt'
```

- FileNotFoundError means exactly what you think it means.

Reading Everything At Once

- The `read` method of a file object returns the entire content of the file as a string.
- Here you can see the lines are **separated** using the **newline** character, **\n**.

```
>>> f = open('colours.txt')
>>> s = f.read()
>>> s
'red\ngreen\nblue\n'
```

Reading Line-By-Line

- Sometimes it is not a good idea to read all of a file at once.
  - For large files this could be **extremely slow**.
  - Very, very large files might **not fit in memory**.
  - In other situations, it might not be convenient to **work** with the **entire** file all at **once**.

- The `readline` method can be used to **read** one line at a **time**.

Reading Line-By-Line

```
>>> f = open('colours.txt')
>>> f.readline()
'red\n'

>>> f.readline()
'green\n'

>>> f.readline()
'blue\n'

>>> f.readline()
''
```

```
red
green
blue

colours.txt
```

- Reading a line will "advance" the cursor position in the file.
- Each line includes the **newline** character at the **end**.
- Once the end of the file is reached, calls to `readline` return the empty string, `''`.

Reading Line-By-Line

- The **newline** character can be removed from lines using **string slicing** or **string replacement**.

```
>>> f = open('colours.txt')
>>> line = f.readline()
>>> line
'red\n'

>>> line[:-1]
'red'

>>> line.replace('\n', '')
'red'
```

Returning to the Start of the File

- You can return the cursor position to the beginning of the file by using the seek file object method and passing 0 in as the argument.

- This allows you to read the same lines multiple times if you wish.

```
>>> f = open('colours.txt')
>>> f.readline()
'red\n'

>>> f.readline()
'green\n'

>>> f.seek(0)
0

>>> f.readline()

'red\n'
```

```
red
green
blue

colours.txt
```

Reading with a For Loop

- For convenience, Python allows you to use a file object in a for loop.
- The file **object** acts like a sequence of lines.
- Behind the scenes, text will be read from the file one line at a time.

Example: Displaying File Contents

```
# File: display_file.py
file_name = input('Enter file name: ')
file = open(file_name)
print('---')
for line in file:
    print(line[:-1])
```

```
$ python display_file.py
Enter file name: colours.txt
---
red
green
blue
```

```
red
green
blue
```

colours.txt

Check Your Understanding

Q. Based on your understanding of how `readline` works, what do you think the result of calling `read` twice in a row on the same file object would be?

Check Your Understanding

Q. Based on your understanding of how `readline` works, what do you think the result of calling `read` twice in a row on the same file object would be?

A. The first call returns the entire contents of the file as a string. The second call returns an empty string.

This is because after the first read call, the cursor position is at the end of the file.

Writing Text Files

File Modes

- When opening a file you can specify a file mode.
- The file mode determines:
  - Whether the file object allows **reading**.
  - Whether the file object allows **writing**.
  - The initial file cursor position.
  - Whether **existing files** should have their contents **kept** or **erased**.
  - Whether the file should be **created** if it doesn't already **exist**.

File Modes

- The default file mode is `'r'`.
- This file mode allows for the file to be read from, but **not** written to.

```
f = open('colours.txt')
```

...is short for...

```
f = open('colours.txt', 'r')
```

Common File Modes

MODE	READ	WRITE	POSITION	KEEP CONTENTS?	CREATE FILE?
`'r'`	✓	✗	Start	✓	✗
`'w'`	✗	✓	Start	✗	✓
`'a'`	✗	✓	End	✓	✓
`'r+'`	✓	✓	Start	✓	✗
`'w+'`	✓	✓	Start	✗	✓
`'a+'`	✓	✓	End	✓	✓

Common File Modes

- For example, when using mode 'a+':
  - It will be possible to both **read** and **write**.
  - If the file **exists**, existing **contents** will be **kept**.
  - If the file **does not** exist, it will be **created**.
  - The **cursor** position will **initially** be at the **end**.

MODE	READ	WRITE	POSITION	KEEP CONTENTS?	CREATE FILE?
`'a+'`	✓	✓	End	✓	✓

Writing Text

- To write to a file, you must **open** it using a file mode which enables writing, such as '**w**'.
  - e.g. `open('results.txt', 'w')`

- You can write to a file using the `write` method.

- When you have finished writing to a file, it should be closed using the `close` method.
  - This ensures that data is saved to the file correctly.

Example: Writing Text

```
f = open('gifts.txt', 'w')
f.write('A gift is nice,\n')
f.write('but two gifts are better!\n')
f.close()
```

Output:

```
A gift is nice,
but two gifts are better!
```

gifts.txt

Notice that we had to include newline characters (\n) to explicitly indicate when we want a new line in the file to start.

This is different from print statements, which automatically add a newline after every message displayed to the user.

Appending Text

- Opening an existing file with file mode 'w' or 'w+' will erase the contents of the file.
  - Be careful! This could cause you to lose data.

- If you want to **keep** the **existing** contents of the file you are writing to, use the file mode '**a**'.
  - The letter 'a' stands for append.
  - Append means "**add to the end**".

Example: Work Tracker

```
# File: work_tracker.py
print('Enter work completed:')
work = input('> ')
work_file = open('work_log.txt', 'w')
work_file.write(work + '\n')

work_file.close()
```

Example: Work Tracker

```
$ python work_tracker.py
Enter work completed:
> Dusted the cupboards
```

```
Dusted the cupboards
```

work_log.txt

Example: Work Tracker

```
$ python work_tracker.py
Enter work completed:
> Dusted the cupboards
```

```
$ python work_tracker.py
Enter work completed:
> Vacuumed the floors
```

```
Vacuumed the floors
```

work_log.txt

Example: Work Tracker

```
# File: work_tracker.py
print('Enter work completed:')
work = input('> ')
work_file = open('work_log.txt', 'w')
work_file.write(work + '\n')

work_file.close()
```

- Every time the program is run, work_log.txt will start as a blank file.
- Any data already in work_log.txt will be erased.

Example: Work Tracker

```
# File: work_tracker2.py
print('Enter work completed:')
work = input('> ')
work_file = open('work_log2.txt', 'a')
work_file.write(work + '\n')

work_file.close()
```

- Here we have changed the file mode to 'a'.
- The first time the program is run, a file called work_log2.txt will be created.
- On each subsequent run, a new line of text will be appended to the file.

Example: Work Tracker

```
$ python work_tracker2.py
Enter work completed:
> Dusted the cupboards
```

```
Dusted the cupboards
```

work_log2.txt

Example: Work Tracker

```
$ python work_tracker2.py
Enter work completed:
> Dusted the cupboards
```

```
$ python work_tracker2.py
Enter work completed:
> Vacuumed the floors
```

```
Dusted the cupboards
Vacuumed the floors
```

work_log2.txt

Check Your Understanding

Q. What will be the contents of `output.txt` after the program finishes running?

```
f = open('output.txt', 'w')
for i in range(5):
    f.write(str(i))

f.close()
```

Check Your Understanding

Q. What will be the contents of `output.txt` after the program finishes running?

```
f = open('output.txt', 'w')
for i in range(5):
    f.write(str(i))
f.close()
```

A.

```
01234
```

output.txt

- range(5) is the sequence 0,1,2,3,4.
- Since the strings that we wrote to the file did not end in a newline, they were all written as part of the same line.

File System Operations

Files and Directories

- There are two types of entries in a computer file system: files and directories.
  - A directory contains other files and/or directories.
  - A file contains data.

- Directories are sometimes called "**folders**".
- The `open` function is for working with files only (not **directories**).

File System Paths

- A path is the **location** of a **file** or **directory**.
  - e.g. on Windows: C:\Program Files\Microsoft Office
  - e.g. on Mac OS: /private/etc/hosts

- Every **file** and **directory** on your computer has a path.

- The first argument to `open` is the path of the file to open.

File System Paths

- An absolute path is **fully-qualified** and globally **unique** on your computer.
- If you create a file using the absolute path "C:\fruit.txt", it doesn't matter **where** you run the program from.
- A relative path is relative to the **current directory**.
- If you create a file using the relative path "fruit.txt", it will be **placed** in the **directory** that you ran the program from.

The `os` Module

- Python provides many functions for working with the **file system** using **paths**.

- Many of these functions are part of the **os** module.

- Use the following line of code to import the os module:

```
import os
```

Identifying Paths

- To see whether a **path refers** to a **file**, use `os.path.isfile(path)`

- **True** will be **returned** if path refers to an **existing** file.

- To see whether a path refers to a directory, use `os.path.isdir(path)`

- **True** will be returned if **path** refers to an **existing** directory.

Example: Identifying Paths

```
# File: identify_path.py
import os
path = input('Enter a path: ')
if os.path.isfile(path):
    print(f'"{path}" is a file.')
elif os.path.isdir(path):
    print(f'"{path}" is a
directory.')
else:
    print(f'"{path}" does not
exist.')
```

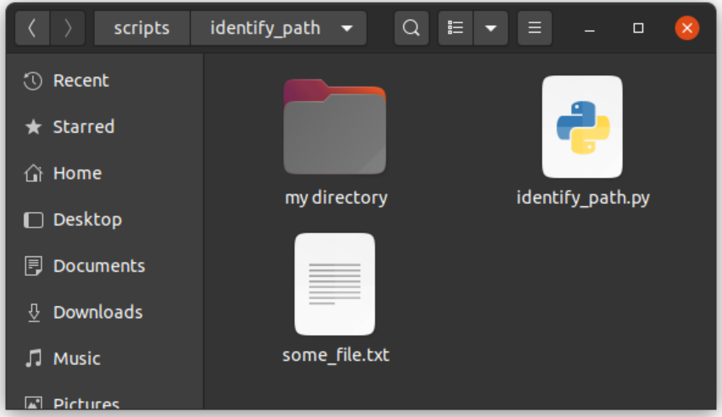
- This program asks the user to input a path, then outputs whether the path refers to a file or directory.
- Works for both absolute and relative paths.

Example: Identifying Paths

```
$ python identify_path.py
Enter a path: my directory
"my directory" is a directory.
```

```
$ python identify_path.py
Enter a path: some_file.txt
"some_file.txt" is a file.
```

```
$ python identify_path.py
Enter a path: nothing
"nothing" does not exist.
```



Creating Directories

- Earlier we saw that files can be created simply by opening them in certain file modes.
- Directories can be created using `os.mkdir(path)`.
- mkdir is short for "make directory".

```
>>> import os
>>> os.path.isdir('my_dir')
False
>>> os.mkdir('my_dir')
>>> os.path.isdir('my_dir')
True
```

Listing Directory Entries

- Finding the **names** of files and directories contained within a directory is called listing a **directory**.
- Listing directories is useful when:
  - **Searching for files**.
  - **Batch processing files**.

- Calling `os.listdir(path)` will return a sequence of file and directory names in the specified directory.
- You can **loop over** these using a for loop.

Renaming Files/Directories

- Calling `os.rename(source_path, dest_path)` will rename the file/directory at source_path to the new name dest_path.

- **Moving** a file is equivalent to **renaming** it.

Removing Files

- Calling `os.remove(path)` will remove the *file* at path.
- This function **cannot** be used to remove directories.

Removing Directories

- Calling `os.rmdir(path)` will remove the *directory* at path.
  - rmdir is short for "remove directory".
- This function can only be used to remove empty directories.

File System Operations Reference Table

FUNCTION	DESCRIPTION	EXAMPLE
`os.path.isfile`	Is an existing file?	`os.path.isfile('book.txt')`
`os.path.isdir`	Is an existing directory?	`os.path.isdir('Songs')`
`os.mkdir`	Create a directory.	`os.path.mkdir('results')`
`os.listdir`	List a directory.	`os.listdir('input_files')`
`os.rename`	Rename/move a file/directory.	`os.rename('old.txt', 'new.txt')`
`os.remove`	Remove a file.	`os.remove('temporary.dat')`
`os.rmdir`	Remove a directory.	`os.rmdir('empty_dir')`

Check Your Understanding

Q. Using the file operations explained in this lecture, how might you go about deleting a directory containing files (but no sub-directories)?

Check Your Understanding

Q. Using the file operations explained in this lecture, how might you go about deleting a directory containing files (but no sub-directories)?

A. You could list the files in the directory, delete those files, and then delete the now empty directory.

```
import os
d = 'my_dir'
for file_name in os.listdir(d):
    p = os.path.join(d, file_name)
    os.remove(p)
os.rmdir(d)
```

Next Lecture We Will...

- Represent and manipulate arbitrary sequences of items using lists.

Thanks for your attention!

The slides and lecture recording will be made available on LMS.

PPT Acknowledgement: Dr Aiden Nibali, CS&IT LTU.