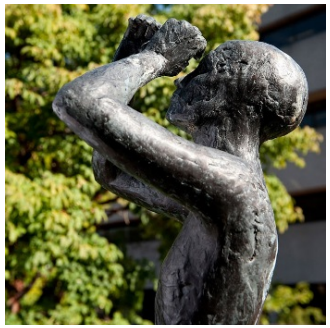


CS1117 – Introduction to Programming

Dr. Jason Quinlan,

School of Computer Science and Information Technology

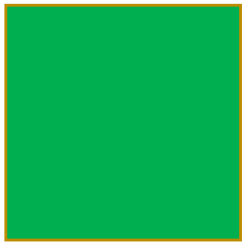
A TRADITION OF INDEPENDENT THINKING

UCC

University College Cork, Ireland

Coláiste na hOllscoile Corcaigh

Semester 1 revision



If any of the content, we cover in these revision lectures is confusing, ask questions

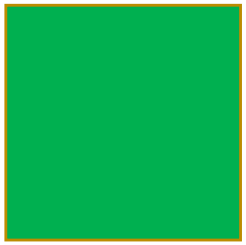
If not in class, ask on the anonymous google form

We will then cover the content in the next class or in the extra coding class

This is your chance to get to know this material

☺

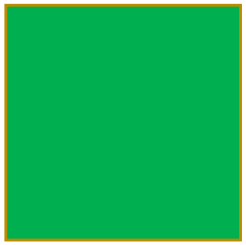
Python Functions



Some of the operators are:

Conversion	Meaning
'd'	Signed integer decimal.
'f'	Floating point decimal format.
'c'	Single character (accepts integer or single character string).
'r'	String (converts any Python object using repr()).
's'	String (converts any Python object using str()).
'a'	String (converts any Python object using ascii()).
'%'	No argument is converted, results in a '%' character in the result.

Semester 1 revision



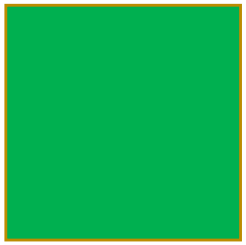
```
s = 'Hello, All.'
print str(s)
print str(2.0/11.0)
```

Output:

```
Hello, All.
0.181818181818
```

str() provides readable content

Semester 1 revision



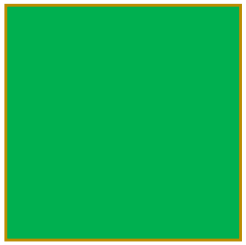
```
s = 'Hello, All.'
print repr(s)
print repr(2.0/11.0)
```

Output:

```
'Hello, All.'
0.18181818181818182
```

repr() provides actual content - representation

String Formatting



In this example - \t adds a tab to our output string

```
print("integer operator %%d on int \t%d" % 7)
print("float operator %%f on float \t%f" % 7.0)
print("integer operator %%d on float \t%d" % 7.0)
print("float operator %%f on int \t%f" % 7)
print("string operator %%s on string \t%s" % "7")
print("string operator %%s on int \t%s" % 7)
print("string operator %%s on float \t%s" % 7.0)

# output
# integer operator %d on int      7
# float operator %f on float      7.000000
# integer operator %d on float    7
# float operator %f on int        7.000000
# string operator %s on string    7
# string operator %s on int       7
# string operator %s on float     7.0
```

String Formatting

String formatting offers a mechanism to add spacing and reduce decimal places

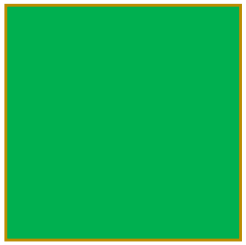
%8.f prints 8 characters - the number, no decimal places and 7 preceding blank spaces

```
print("float   operator %%f on float \t%f" % 7.0)
print("float   operator %%f on float \t%.f" % 7.0)
print("float   operator %%f on float \t%.2f" % 7.0)
print("float   operator %%f on float \t%8.f" % 7.0)

# output
# float   operator %f on float    7.000000
# float   operator %f on float    7
# float   operator %f on float    7.00
# float   operator %f on float           7
```

String Formatting Recap

- We looked at how to import functions from Pythons libraries
- We look at various ways of passing parameters to the print() function
- We saw that string objects have their own set of functions we can calling using the dot (.) operator
- We saw some of the % operators we can use to format input to string objects
- We saw how we can use \t (tab) and \n (newline) to modify the structure of the string output
- And we saw 3 different ways to print a blank line in Python
- Finally, we saw how to create tables in the print output, using numbers in %f

print()

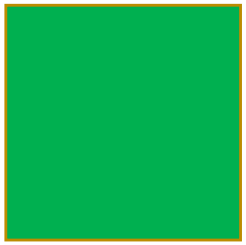


This is the docString for Python's print() function

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file: a file-like object (stream); defaults to the current sys.stdout.
sep: string inserted between values, default a space.
end: string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
```

First thing to note is, it is a very detailed docString and you can see immediately what extra parameters print() has and what their respective role/type() are

print()

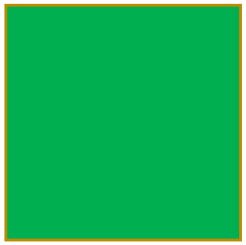


Let's look at "sep" first

sep changes how print() joins the different strings together

```
print("there is a", "in this line")
print("there is a", "in this line", sep=" - ")
print("there is a", "in this line", sep=" word ")

# output
# there is a in this line
# there is a - in this line
# there is a word in this line
```

print()



Here we set the value of end to equal the empty string

```
print("This line stops here")
print("This line runs over ", end="")
print("two lines")

# output
# This line stops here
# This line runs over two lines
```

Semester 1 revision

Week 3

Lecture 7

UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

if statement

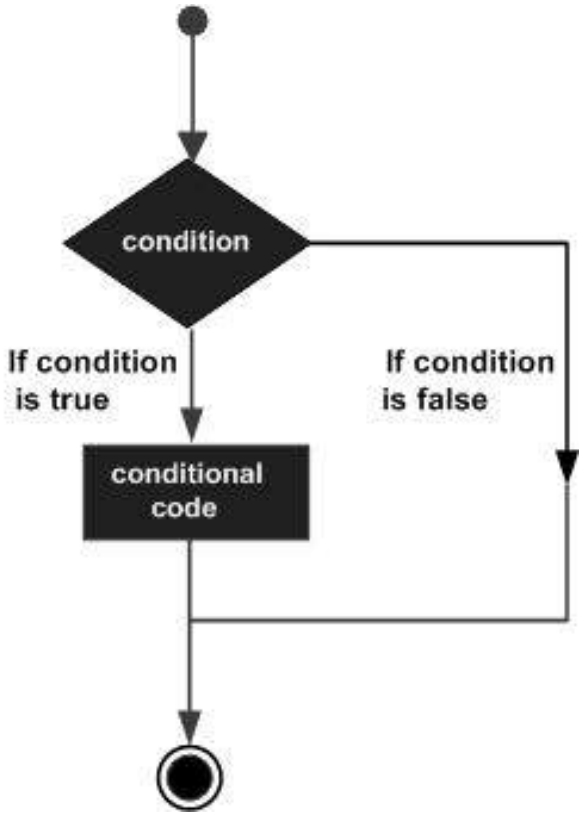
if statements can be modeled as a flow chart.

If the condition is evaluated as true then execute the conditional code (statement block).

Otherwise skip that code.

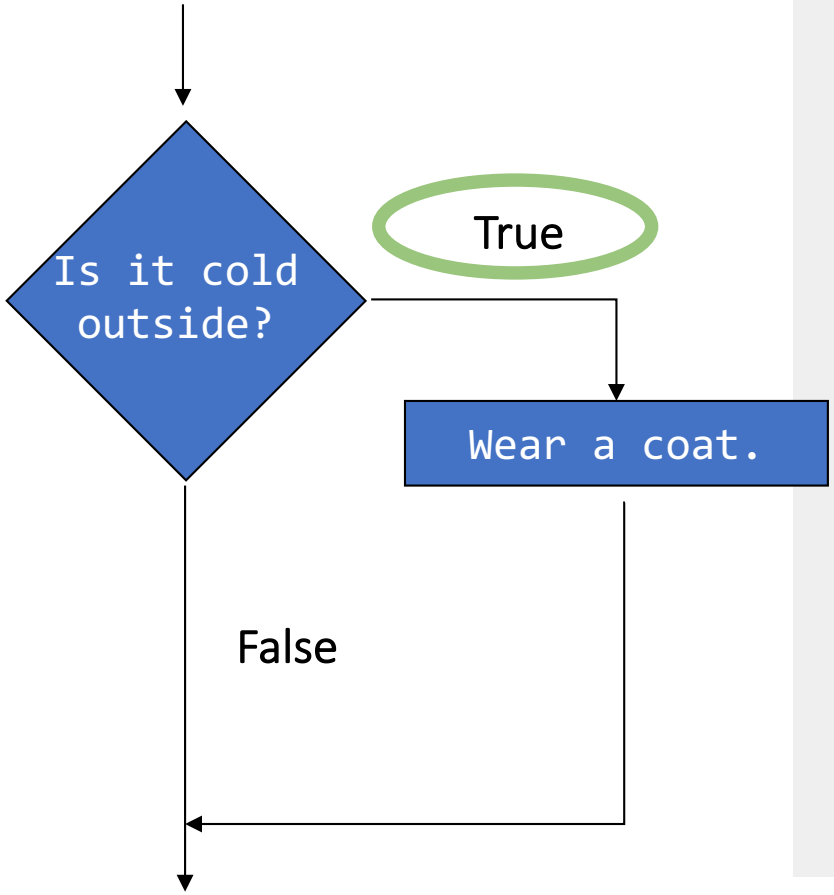
So, every if statement equates to either True or False

That's all…



if statement

```
cold_outside = True

if cold_outside:
    print("wear coat")

# output
# wear coat
```

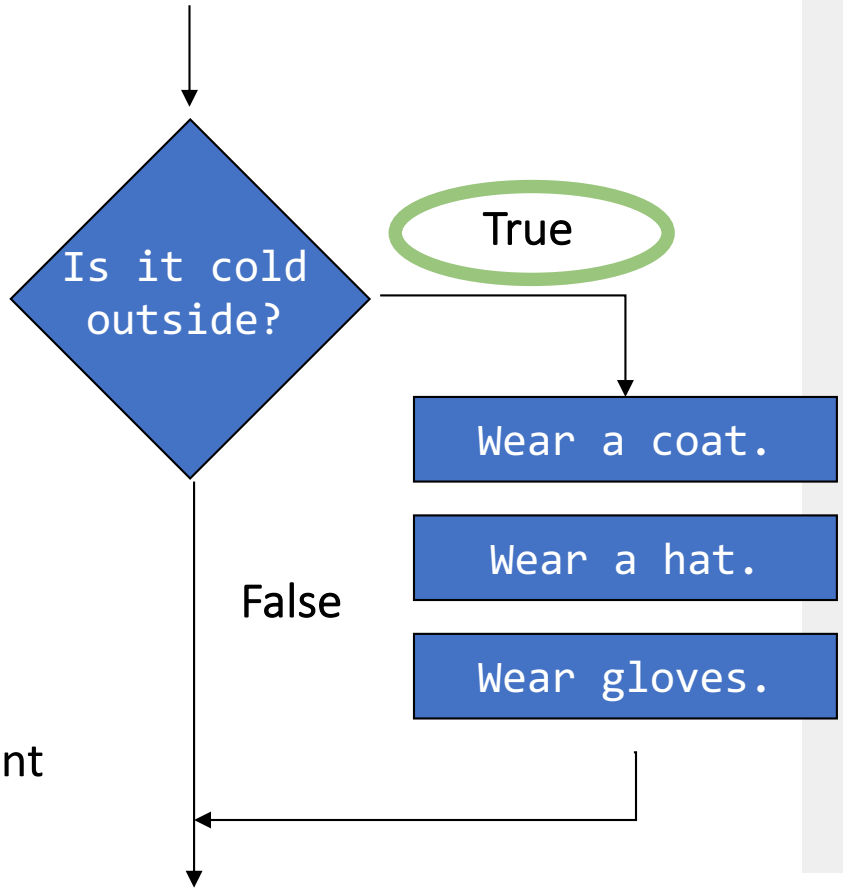


Whatever sits between the if and :

Must equate to a True or False

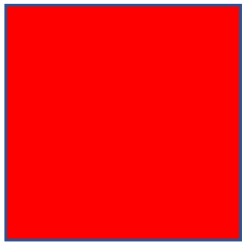
if statement

```
cold_outside = True

if cold_outside:
    print("wear coat")
    print("wear hat")
    print("wear gloves")

# output
# wear coat
# wear hat
# wear gloves
```



Like Functions, we can create statement blocks within the indented code…

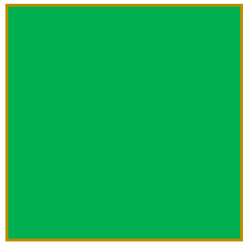
if statement



A condition is also called a *boolean expression* and is any variable or calculation that results in a `True` or `False` condition.

Expression	Meaning
`x > y`	Is x greater than y?
`x < y`	Is x less than y?
`x >= y`	Is x greater than or equal to y?
`x <= y`	Is x less than or equal to y.
`x == y`	Is x equal to y?
`x != y`	Is x not equal to y?

if statement



Python likes to use readable code where possible, so we can

```
if (num_demogorgan != 1):
    print("It's Stranger Things season 1, Eleven will save us")
```

Change the ‘not equal to (!=)’ to ‘equal to (==)’

```
if (num_demogorgan == 1):
    print("It's Stranger Things season 1, Eleven will save us")
```

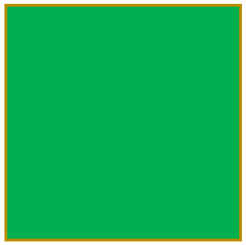
And place not outside the brackets

```
if not (num_demogorgan == 1):
    print("It's Stranger Things season 1, Eleven will save us")
```

And if you even remove the brackets, the output stays the same

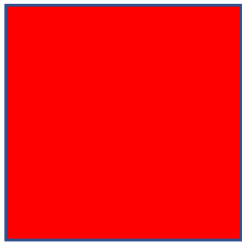
```
if not num_demogorgan == 0:
    print("It's Stranger Things season 1, Eleven will save us")
```

if statement



- The addition of and and not make the code much easier to read
- not will negate the output of the condition
  - So if the condition is True
  - cold_outside is True
  - not cold_outside is equal to False
- and mandates that all the conditions must be True
  - cold_outside is True
  - raining_outside is True
  - if cold_outside and raining_outside are True, the condition is True

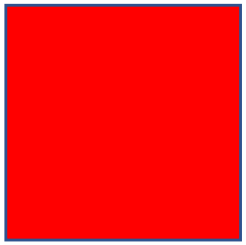
and



For and all expressions (conditions) must be True

Expression 1	Expression 2	Expression1 || Expression2
true	true	true
true	false	false
false	true	false
false	false	false

and



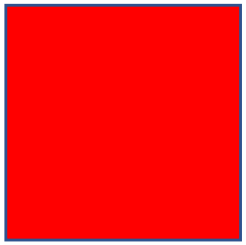
For and all expressions (conditions) must be True

If one or both condition(s) are False

The result is False

Expression 1	Expression 2	Expression1 || Expression2
true	true	true
true	false	false
false	true	false
false	false	false

and



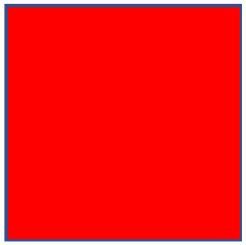
For and all expressions (conditions) must be True

If one or both condition(s) are False

The result is False

Expression 1	Expression 2	Expression1 || Expression2
true	true	true
true	false	false
false	true	false
false	false	false

and



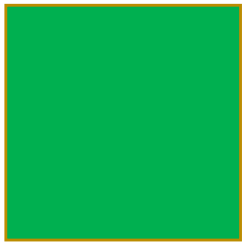
For and all expressions (conditions) must be True

If one or both condition(s) are False

The result is False

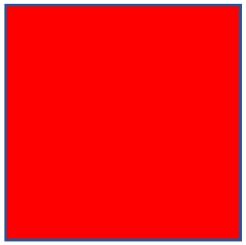
Expression 1	Expression 2	Expression1 || Expression2
true	true	true
true	false	false
false	true	false
false	false	false

if statement



- The addition of and and not make the code much easier to read
- not will negate the output of the condition
  - So if the condition is True
  - cold_outside is True
  - not cold_outside is equal to False
- and mandates that all the conditions must be True
  - cold_outside is True
  - raining_outside is True
  - if cold_outside and raining_outside are True, the condition is True
- and and not are known as Boolean operators
  - They produce a value that can have at most 2 values
- We have one more Boolean operator
  - or mandates if one condition or the other condition is True
  - The entire condition is True

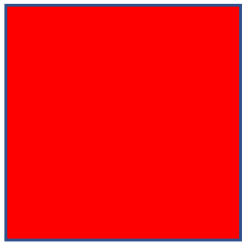
or



For or only one expressions (conditions) needs to be True

Expression 1	Expression 2	Expression1 || Expression2
true	true	true
true	false	true
false	true	true
false	false	false

or

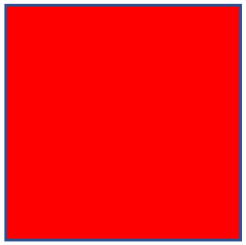


For or only one expressions (conditions) needs to be True

If one or both condition(s) are True -> the result is True

Expression 1	Expression 2	Expression1 || Expression2
true	true	true
true	false	true
false	true	true
false	false	false

or

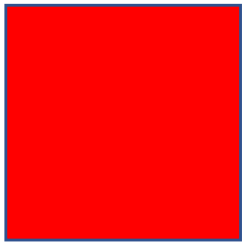


For or only one expressions (conditions) needs to be True

If one or both condition(s) are True -> the result is True

Expression 1	Expression 2	Expression1 || Expression2
true	true	true
true	false	true
false	true	true
false	false	false

or

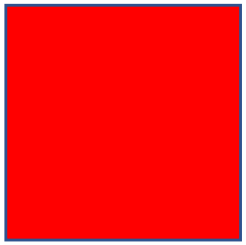


For or only one expressions (conditions) needs to be True

If one or both condition(s) are True -> the result is True

Expression 1	Expression 2	Expression1 || Expression2
true	true	true
true	false	true
false	true	true
false	false	false

or

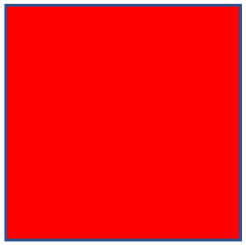


For or only one expressions (conditions) needs to be True

If one or both condition(s) are True -> the result is True

Expression 1	Expression 2	Expression1 || Expression2
true	true	true
true	false	true
false	true	true
false	false	false

or



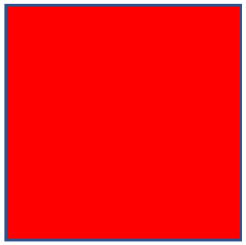
For or only one expressions (conditions) needs to be True

If one or both condition(s) are True -> the result is True

If both condition(s) are False -> the result is False

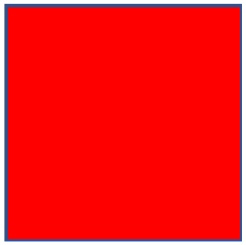
Expression 1	Expression 2	Expression1 || Expression2
true	true	true
true	false	true
false	true	true
false	false	false

String comparison



- When we compare strings using == and !=
- We are actually comparing the string ASCII values

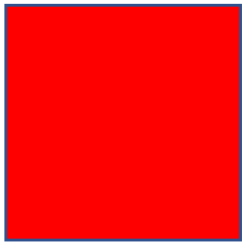
String comparison



- When we compare strings using == and !=
- We are actually comparing the string ASCII values
- **ASCII** stands for American Standard Code for Information Interchange.

- Each character is assigned a unique ASCII value
- ‘A’ (65) has a lower value than ‘Z’ (90)
- ‘A’ (65) and ‘a’ (97) are not the same

String comparison



- When we compare strings using == and !=
- We are actually comparing the string ASCII values
- **ASCII** stands for American Standard Code for Information Interchange.

- Each character is assigned a unique ASCII value
- ‘A’ (65) has a lower value than ‘Z’ (90)
- ‘A’ (65) and ‘a’ (97) are not the same

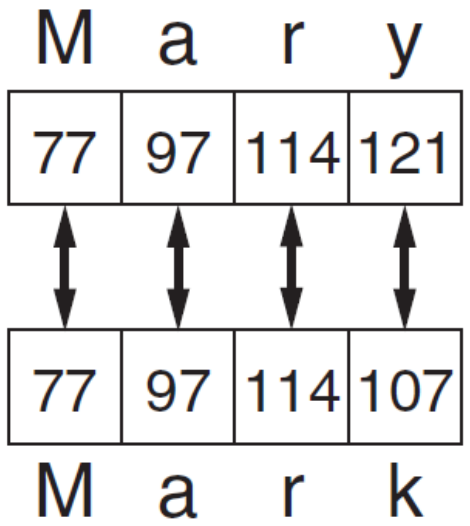
- Let’s look at an example:

String comparison

- When we compare strings using == and !=
- We are actually comparing the string ASCII values
- **ASCII** stands for American Standard Code for Information Interchange.

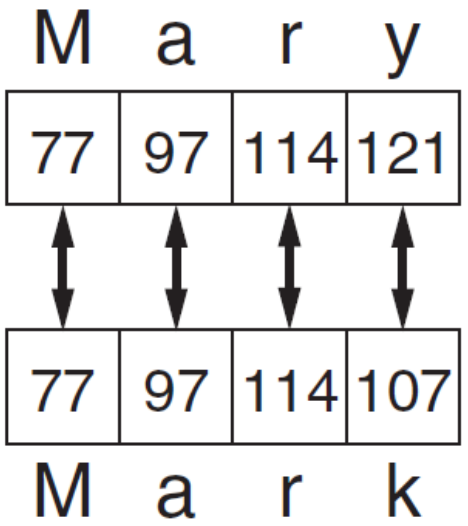
- Each character is assigned a unique ASCII value
- ‘A’ (65) has a lower value than ‘Z’ (90)
- ‘A’ (65) and ‘a’ (97) are not the same

- Let’s look at an example:



String comparison

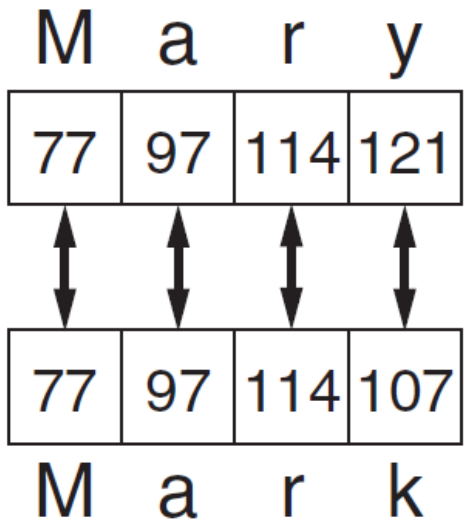
- Mary = 77, 97, 114, and 121
- Mark = 77, 97, 114, and 107



String comparison

- Mary = 77, 97, 114, and 121
- Mark = 77, 97, 114, and 107

```
print("M has the ascii value:", ord('M'))
print("a has the ascii value:", ord('a'))
print("r has the ascii value:", ord('r'))
print("y has the ascii value:", ord('y'))

# output
# M has the ascii value: 77
# a has the ascii value: 97
# r has the ascii value: 114
# y has the ascii value: 121
```

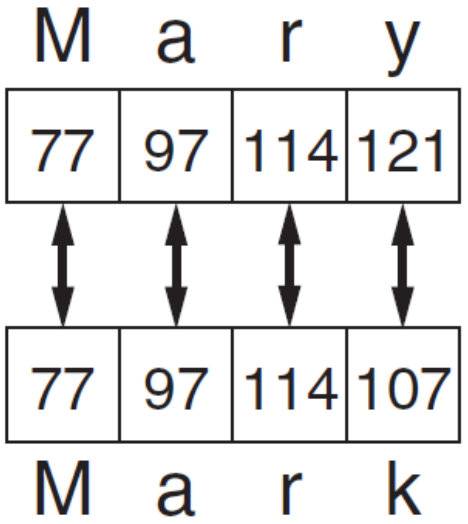


String comparison

- Mary = 77, 97, 114, and 121
- Mark = 77, 97, 114, and 107

```
print("M has the ascii value:", ord('M'))
print("a has the ascii value:", ord('a'))
print("r has the ascii value:", ord('r'))
print("y has the ascii value:", ord('y'))

# output
# M has the ascii value: 77
# a has the ascii value: 97
# r has the ascii value: 114
# y has the ascii value: 121
```

We can use ord() to view the ASCII Unicode value for a single character

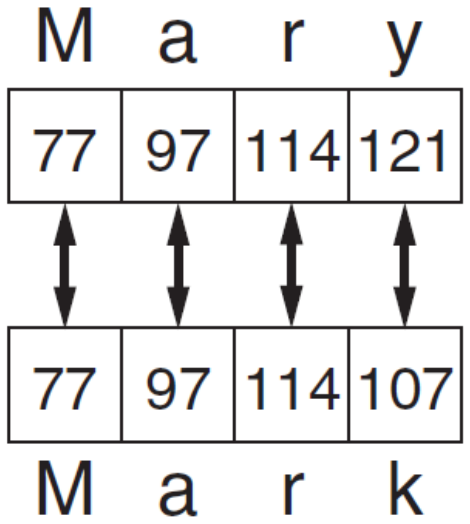


String comparison

- Mary = 77, 97, 114, and 121
- Mark = 77, 97, 114, and 107

- Mark has a lower value than Mary so they are not the same

- As Mark is lower than Mary we can now also use



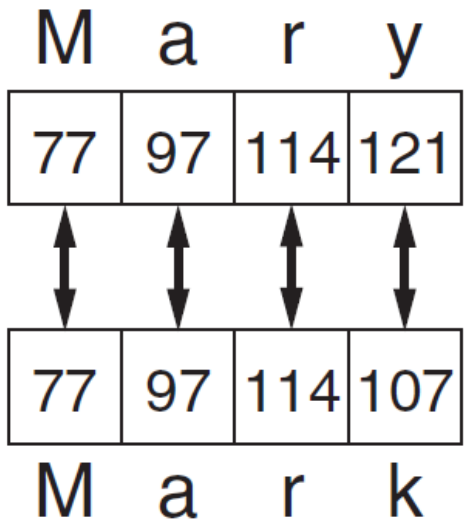
String comparison

- Mary = 77, 97, 114, and 121
- Mark = 77, 97, 114, and 107

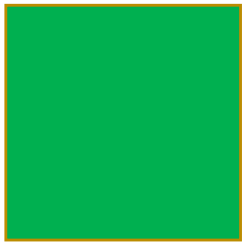
- Mark has a lower value than Mary so they are not the same

- As Mark is lower than Mary we can now also use

- <
- >
- <=
- >=

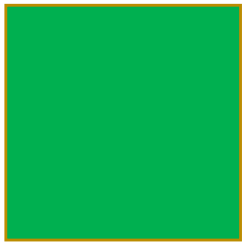


String comparison



- One final comment on string comparison
- If you do want to make sure two values are the same object, i.e., same id()

String comparison



- One final comment on string comparison
- If you do want to make sure two values are the same object, i.e., same id()
- You can use is

- This checks to make sure the underlying objects are the same

String comparison

- One final comment on string comparison
- If you do want to make sure two values are the same object, i.e., same id()
- You can use is

- This checks to make sure the underlying objects are the same

```
name = "__main__"

if name is "__main__":
    print("I'm main")
    print(id(name))
    print(id("__main__"))
    print(name)
    print("__main__")

# output
# I'm main
# 4449166448
# 4449166448
# __main__
# __main__
```

String comparison

- One final comment on string comparison
- If you do want to make sure two values are the same object, i.e., same id()
- You can use is
- This checks to make sure the underlying objects are the same
- Python will give you a warning when you use this ☺

```
name = "__main__"

if name is "__main__":
    print("I'm main")
    print(id(name))
    print(id("__main__"))
    print(name)
    print("__main__")

# output
# I'm main
# 4449166448
# 4449166448
# __main__
# __main__
```

Recap

- We introduced comparison statements
  - if – allows us to check if a condition is True or False
- if is constructed similar to functions
  - if condition:

    indent – statement block of code
- We introduced relational operators
  - < <= == != > >=
  - Permits comparison of different values (variables)
- We introduced Boolean operators
  - and – both expressions must be True for condition to be True
  - not – only one expression must be True for condition to be True
  - or – negates the expression/condition
    - From True to False, or False to True
- We saw how if compares Strings
  - Using ASCII characters (not via object values)
- We can use is to compare Strings using object values

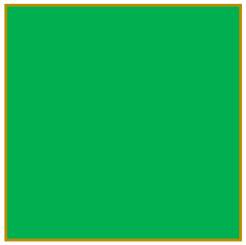
Semester 1 revision

Week 3

Lecture 8

UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

if statement



Let's look at an example…

```
def the_choice(character):

    if character == "Neo":
        print("Neo is my favourite character.")
    elif character == "Trinity":
        print("Trinity is my favourite character.")
    else:
        print("Morpheus is my favourite character.")


the_choice("Trinity")

# output
# Trinity is my favourite character.
```

if, if/else and elif Recap

- We introduced if conditional statements over ranges of values
  - num_demodog from 1 to 10
- We added checks to run code when a conditional statement is False

```
if (condition):
    run code if condition is True
else:
    run code if condition is False
```

- And we added checks for multiple inputs using elif

```
if (condition1):
    run code if condition1 is True
elif (condition2):
    run code if condition2 is True
else:
    run code if both condition1 and condition2 are False
```

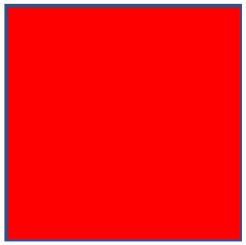
Semester 1 revision

Week 3

Lecture 9

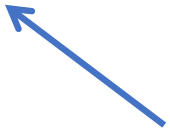
UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Exception Handling



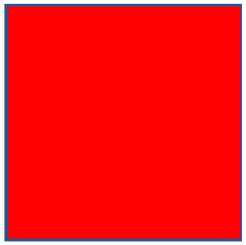
- We see lots and lots of errors when we code, e.g.,

- TypeError: can only concatenate str (not "int") to str



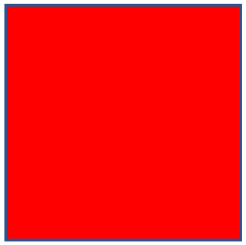
e.g., we have tried to print an int value that we have not cast to a string

Exception Handling

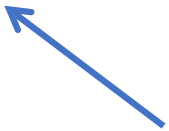


- We see lots and lots of errors when we code, e.g.,
- TypeError: can only concatenate str (not "ValueError") to str
- ValueError: invalid literal for int() with base 10: ‘g’

Exception Handling

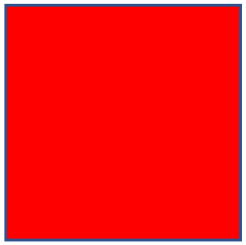


- We see lots and lots of errors when we code, e.g.,
- TypeError: can only concatenate str (not "ValueError") to str
- ValueError: invalid literal for int() with base 10: ‘g’



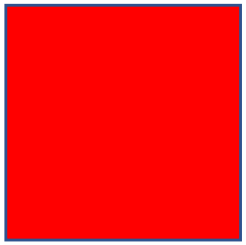
e.g., we have tried to cast an input string to an int, but it is not a int

Exception Handling

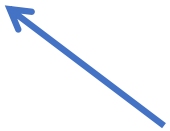


- We see lots and lots of errors when we code, e.g.,
- TypeError: can only concatenate str (not "ValueError") to str
- ValueError: invalid literal for int() with base 10: ‘g’
- NameError: name 'number' is not defined

Exception Handling

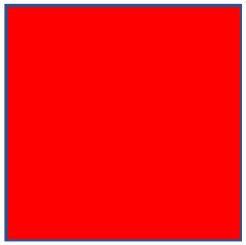


- We see lots and lots of errors when we code, e.g.,
- TypeError: can only concatenate str (not "ValueError") to str
- ValueError: invalid literal for int() with base 10: ‘g’
- NameError: name 'number' is not defined



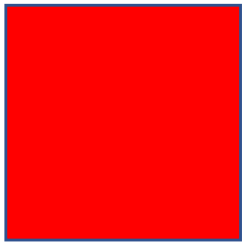
e.g., we have tried to call a variable but we have not yet assigned it a value

Exception Handling



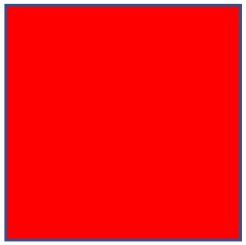
- We see lots and lots of errors when we code, e.g.,
- TypeError: can only concatenate str (not "ValueError") to str
- ValueError: invalid literal for int() with base 10: ‘g’
- NameError: name 'number' is not defined
- Each of these errors are bugs in our code, and while we will get lots of them as we code, we need to consider each and every one of them as we write

Exception Handling



- We see lots and lots of errors when we code, e.g.,
- TypeError: can only concatenate str (not "ValueError") to str
- ValueError: invalid literal for int() with base 10: ‘g’
- NameError: name 'number' is not defined
- Each of these errors are bugs in our code, and while we will get lots of them as we code, we need to consider each and every one of them as we write
- But….

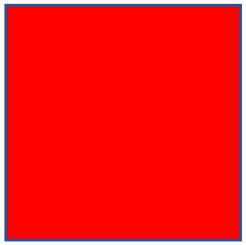
Exception Handling



- Python give us a very simple mechanism to catch these and all other errors:

- Exception Handling:

Exception Handling

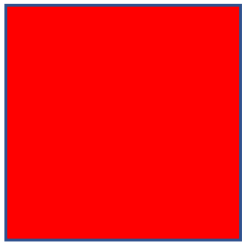


- Python give us a very simple mechanism to catch these and all other errors:

- Exception Handling:

- Instead of crashing, the exception handler prints a message indicating that there was a problem, or we can run some code.

Exception Handling



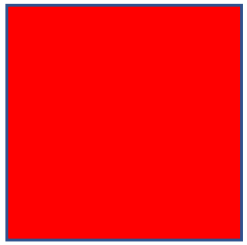
- Python give us a very simple mechanism to catch these and all other errors:

- Exception Handling:

- Instead of crashing, the exception handler prints a message indicating that there was a problem, or we can run some code.

- The `try…except` can be used to catch *any* kind of error and provide for a graceful exit.

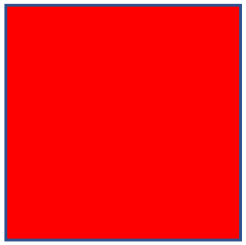
Exception Handling



The programmer can write code that catches and deals with errors that arise while the program is running, i.e., “Do these steps, and if any problem crops up, handle it this way.”

```
try:
      <body>
except:
      <handler>
```

Exception Handling



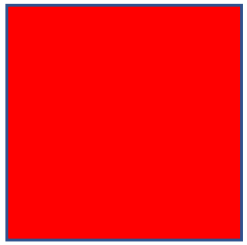
The programmer can write code that catches and deals with errors that arise while the program is running, i.e., “Do these steps, and if any problem crops up, handle it this way.”

```
try:
      <body>
except:
      <handler>
```

When Python encounters a try statement, it attempts to execute the statements inside the body.

If there is no error, control passes to the next statement after the try…except.

Exception Handling

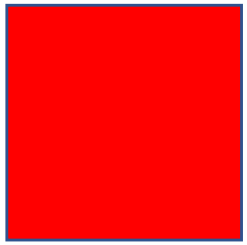


Quick query:

```
number = input("Please enter a positive number: ")

try:
    # add back the int check
    number = int(number)
    # check if number is negative
    if number <= 0:
        print(str(number) + " is negative, please try again with a positive number")
    # if positive and even
    elif number % 2 == 0:
        print(str(number) + " is even")
    else:
        print(str(number) + " is odd")
    # if not positive, print error message
except Exception as e:
    print(number + " is not a number, please retry with a positive number")
    print("the error was: \"" + str(e) + "\"")
```

Why is the int check inside the try?

Exception Handling

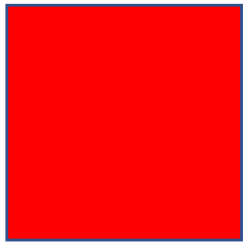


Let’s go back to the original code, and let’s look at some output

```
number = input("Please enter a positive number: ")

try:
    # add back the int check
    number = int(number)
    # check if number is negative
    if number <= 0:
        print(str(number) + " is negative, please try again with a positive number")
    # if positive and even
    elif number % 2 == 0:
        print(str(number) + " is even")
    else:
        print(str(number) + " is odd")
    # if not positive, print error message
except Exception as e:
    print(number + " is not a number, please retry with a positive number")
    print("the error was: \"" + str(e) + "\"")
```

```
Please enter a positive number: h
h is not a number, please retry with a positive number
the error was: "invalid literal for int() with base 10: 'h'"
```

Exception Handling

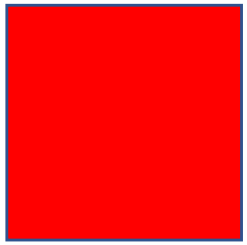


Let’s go back to the original code, and let’s look at some output

```
number = input("Please enter a positive number: ")

try:
    # add back the int check
    number = int(number)
    # check if number is negative
    if number <= 0:
        print(str(number) + " is negative, please try again with a positive number")
    # if positive and even
    elif number % 2 == 0:
        print(str(number) + " is even")
    else:
        print(str(number) + " is odd")
    # if not positive, print error message
except Exception as e:
    print(number + " is not a number, please retry with a positive number")
    print("the error was: \"" + str(e) + "\"")
```

```
Please enter a positive number: 9.0
9.0 is not a number, please retry with a positive number
the error was: "invalid literal for int() with base 10: '9.0'"
```

Exception Handling

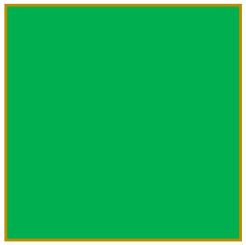


Let's go back to the original code, and let's look at some output

```
number = input("Please enter a positive number: ")

try:
    # add back the int check
    number = int(number)
    # check if number is negative
    if number <= 0:
        print(str(number) + " is negative, please try again with a positive number")
    # if positive and even
    elif number % 2 == 0:
        print(str(number) + " is even")
    else:
        print(str(number) + " is odd")
    # if not positive, print error message
except Exception as e:
    print(number + " is not a number, please retry with a positive number")
    print("the error was: \"" + str(e) + "\"")
```

```
Please enter a positive number: (3,6,8)
(3,6,8) is not a number, please retry with a positive number
the error was: "invalid literal for int() with base 10: '(3,6,8)'"
```

Tuple

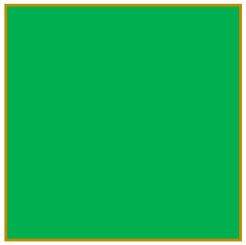


- We've seen Tuple a few times so far in this class
- Tuple creation and assignment:
  - x = ("Ed", "Edd", "Eddy", 2009)

- Functions that return more than one value:
  - print(average_and_modulus_of_two(2,4))
  - # output => (3,0)

- Using the string partition function:
  - print("hello world".partition("w"))
  - # output => ('hello ', 'w', 'orld')

Tuple



As previously stated:

A tuple is an unordered collection of values

A tuple is immutable (contents cannot be changed)

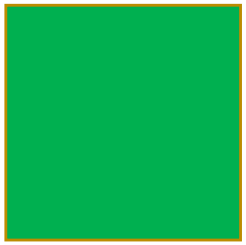
A tuple is created using ”round brackets”

```
x = ("Ed", "Edd", "Eddy", 2009)
```

A tuple can contain a mixture of variable types

e.g., int, float, string, etc.

Tuple

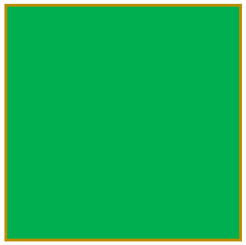


Let’s look at some output

Assign and print a tuple – option 1

```
great_show = ("Ed", "Edd", "Eddy", 2009)
print(great_show)

# output
# ('Ed', 'Edd', 'Eddy', 2009)
```

Tuple



Let’s look at some output

We can print individual values in the tuple

```
great_show = ("Ed", "Edd", "Eddy", 2009)
print(great_show[1])

# output
# Edd
```

Tuple

Let’s look at some output

We can print individual values in the tuple

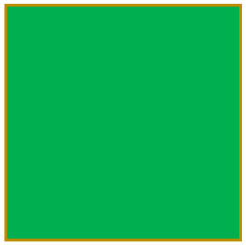
```
great_show = ("Ed", "Edd", "Eddy", 2009)
print(great_show[1])

# output
# Edd
```

We select index 1

“Edd” is printed….

Remember 0 indexing in CS

Tuple

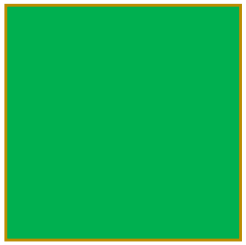


Let’s look at some output

We can also get the number of items in the tuple

```
great_show = ("Ed", "Edd", "Eddy", 2009)
print("There are "+str(len(great_show))+" items in this tuple")

# output
# There are 4 items in this tuple
```

Tuple

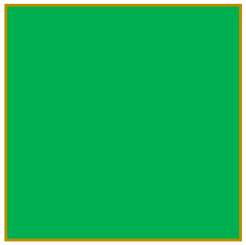


Let’s look at some output

if we add a new item to the tuple

```
great_show = ("Ed", "Edd", "Eddy", 2009)
great_show[4] = "plank"

# output
# TypeError: 'tuple' object does not support item assignment
```

Tuple



Let’s look at some output

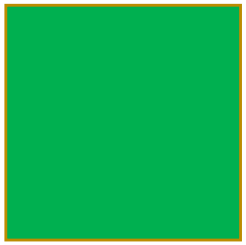
if we add a new item to the tuple

```
great_show = ("Ed", "Edd", "Eddy", 2009)
great_show[4] = "plank"

# output
# TypeError: 'tuple' object does not support item assignment
```

Using square brackets, we add “plank” to index 4

We get an error – tuple does not support assignment

Tuple



Tuple has two other commonly used functions

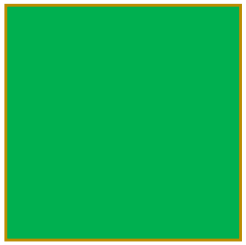
count()

```
great_show = ("Ed", "Ed", "Ed", 2009)
print(str(great_show[0]) + " occurs "
      + str(great_show.count(great_show[0]))
      + " time(s) in this tuple")

# output
# Ed occurs 3 time(s) in this tuple
```

count() returns the number of times

a value appears in a tuple

Tuple



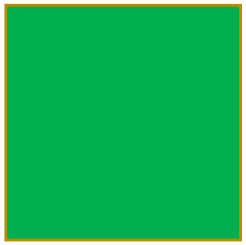
Tuple has two other commonly used functions

index()

```
great_show = ("Ed", "Ed", "Ed", 2009)
print("Ed occurs at index "
      + str(great_show.index("Ed"))
      + " in this tuple")

# output
# Ed occurs at index 0 in this tuple
```

index() returns the first index position that a value appears in a tuple

Tuple



Let's look at some output

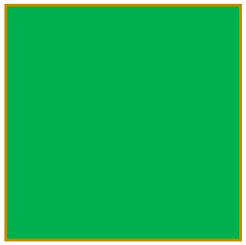
What happens when we check for a value not in the list

```
great_show = ("Ed", "Ed", "Ed", 2009)
print("ed occurs at index "
      + str(great_show.index("ed"))
      + " in this tuple")

# output
# ValueError: tuple.index(x): x not in tuple
```

We now get an error – ValueError

This is no good to us, as this stops the program, so…

Tuple



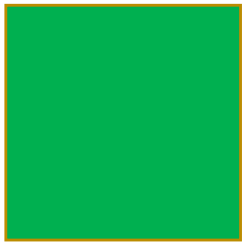
Let’s look at some output

Let’s add some exception handling - a try/except

```
try:
    great_show = ("Ed", "Ed", "Ed", 2009)
    print("ed occurs at index "
          + str(great_show.index("ed"))
          + " in this tuple")
except Exception as e:
    print("ed does not occurs in "
          + str(great_show))

# output
# ed does not occurs in ('Ed', 'Ed', 'Ed', 2009)
```

Now we get a print statement telling us there is a problem

Tuple



So after 40 slides on tuples, I get to the point I want to make

Because we can now determine if a value is in a tuple

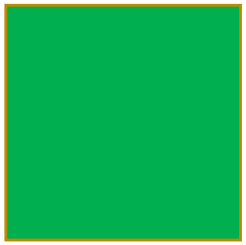
We can write

if value in tuple:

And this will return a True or False

This is known as membership

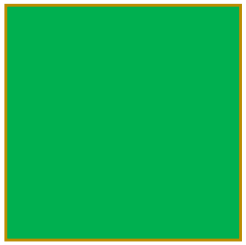
Tuple



if value in tuple:

```
great_show = ("Ed", "Ed", "Ed", 2009)
value_to_find = "ed"

if value_to_find in great_show:
    print(value_to_find + " occurs at index "
          + str(great_show.index(value_to_find))
          + " in this tuple")
else:
    print(value_to_find + " does not occurs in "
          + str(great_show))

# output
# ed does not occurs in ('Ed', 'Ed', 'Ed', 2009)
```

Tuple recap



if value in tuple:

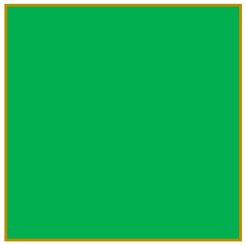
So we can use if to find out if a value is in a tuple

And we do not need to use try/except

This you can use for if/elif/else

Cool ☺

Tuple recap



Oh and if we can use

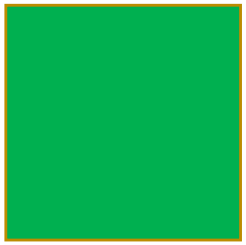
```
if value in tuple:
```

We can also use:

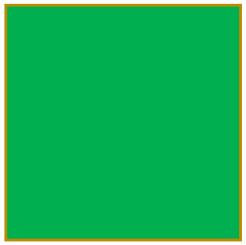
```
if value not in tuple:
```

To negate an input, without using the else

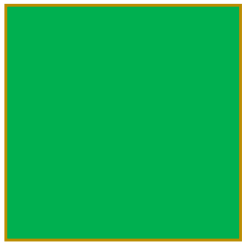
Semester 1 revision



Semester 1 revision



Semester 1 revision



UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh