MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Computer Science Department

SOFT6007 – Web Development Fundamentals

Week 6 – Functions

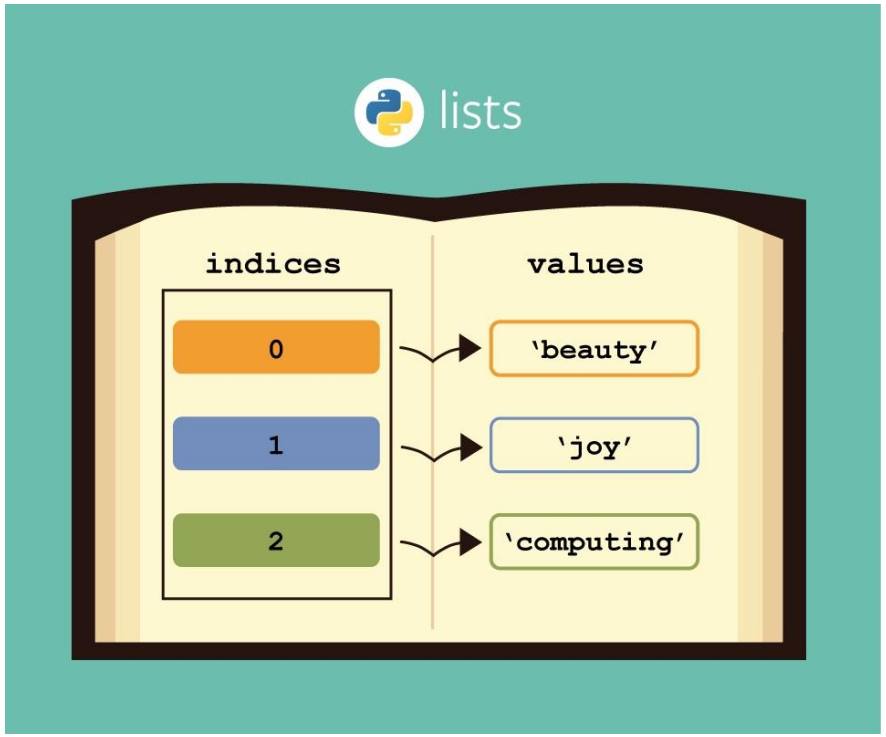
Dr. Bruno Andrade

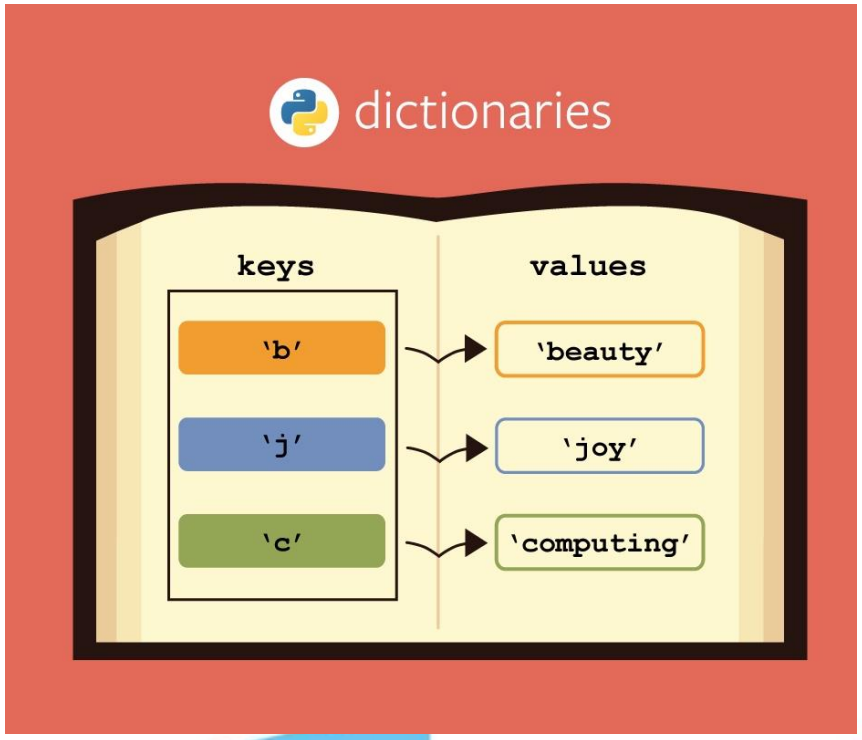
October, 2025

www.mtu.ie

Last week

- Data Structures.
- Lists, Tuples, Dictionaries and Sets:





MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

- **Summary:**
  - Introduction to Functions
  - Defining and Calling a Function
  - Local Variables
  - Global Variables and Global Constants
  - Value-Returning functions
  - Type Annotations
  - Lambda functions
  - Docstrings
  - Opening and saving stuff with Python

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Introduction to Functions

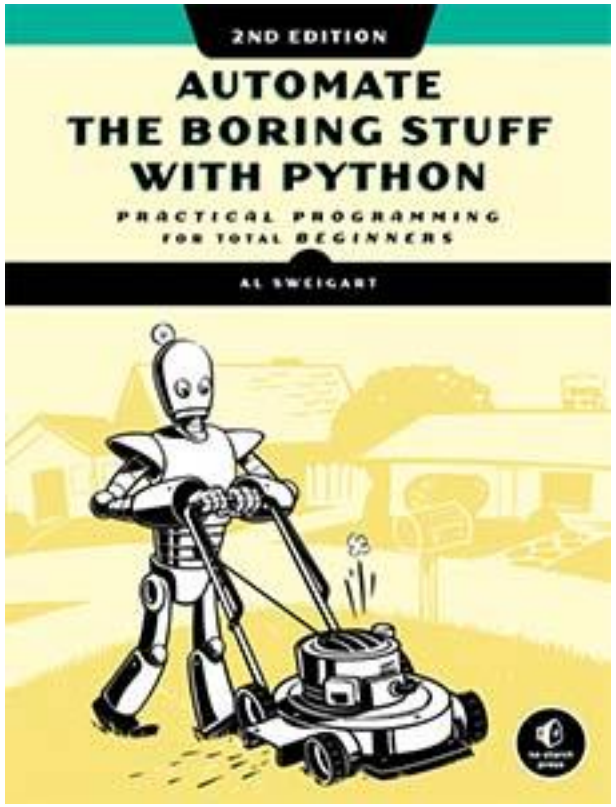
Introduction to Functions

- Up until now you guys learned lots of Python stuff.
- After weeks of programming and, sometimes doing the same processes (but slightly different) you probably realised…

Introduction to Functions

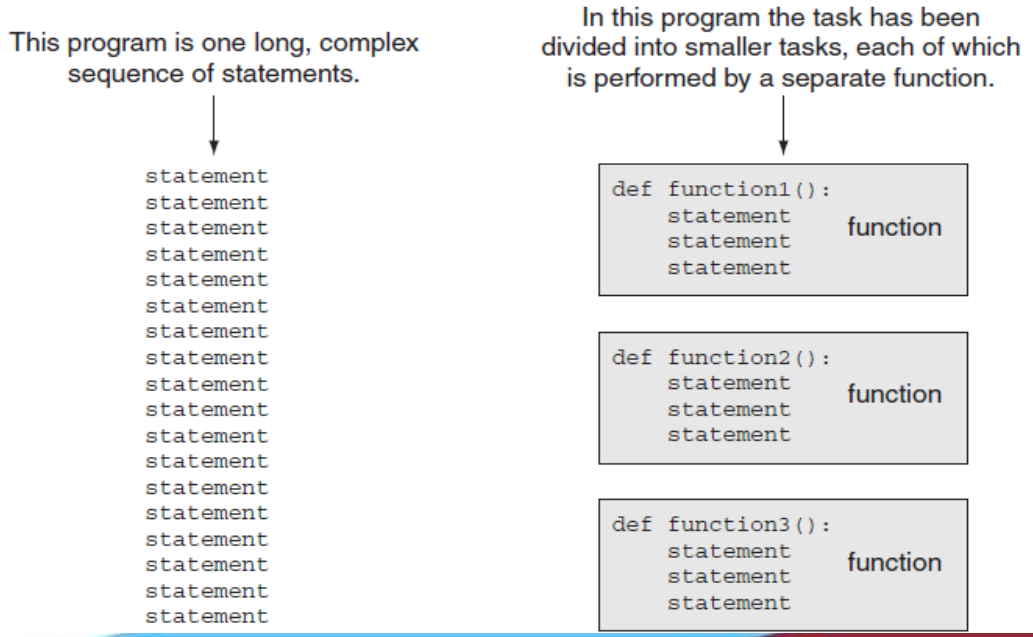
- Programming is about automating processes.
- Functions exist because of this, they allow you to re-use a piece of code anywhere.

```
def Function():
    Piece of code
```



Introduction to Functions

- Most programs perform tasks that are large enough to be broken down into **several subtasks**.
- For this reason, programmers usually break down their programs into functions, to make it more manageable (and smaller).



Benefits of Modularizing a Program with Functions

- The benefits of using functions include:
  - More organized code
  - Easier to read
  - Code reuse
    - Write the code once and call it multiple times
  - Better testing and debugging
    - Can test and debug each function individually
  - Easier facilitation of teamwork
    - Different team members can write different functions
    - Faster development

Defining and Calling a Function

- Function name should be **descriptive** of the task carried out by the function.

- Function definition: specifies what a function does

```
def function_name():
        statement
        statement
```

Function header: first line of function.
Includes keyword **def** and **function name**, followed by parentheses and colon

Under the function header will appear an **indented** block of statements that belong together as a group

Blank lines that appear in a block are **ignored**

Defining and Calling a Function

```
def helloWorld():
    print ('hello world')

helloWorld()
```

```
hello world
```

Defining and Calling a Function

- It is common for a program to have a **main** function that is called when the program starts.

- `main` function: called when the program starts
  - Calls other functions when they are needed

```
def helloWorld():
    print ("hello world")
def main():
    helloWorld()
    print ("GoodBye")
main()
```

```
hello world
GoodBye
```

The main function is called first. The main function then calls the helloWorld function.

The interpreter then returns to the main function.

Positioning of Functions

- The following code will **not work**. It will produce the following error:
  - NameError: name 'main' is not defined

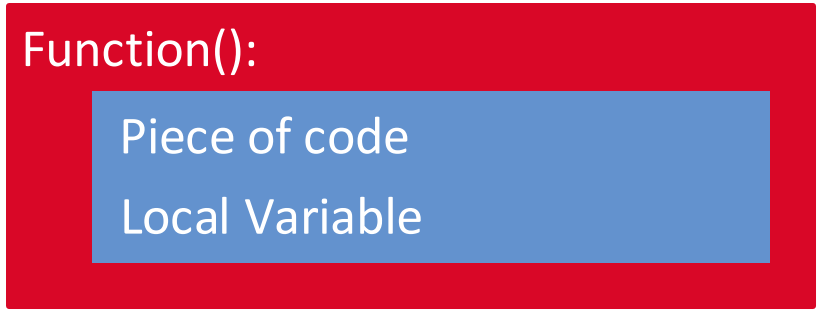
```
main()

def helloWorld():
    print ("hello world")

def main():
    helloWorld()
    print ("GoodBye")
```

```
---------------------------------------
---------------------------------------
NameError
Traceback (most recent call last)
/var/folders/9x/hx0451rj4n1fcxsm1m6ch5h
x7p3m0h/T/ipykernel_18133/542690968.py
in <module>
----> 1 main()
      2
      3 def main():
      4     helloWorld()
      5     print ("GoodBye")

NameError: name 'main' is not defined
```

Local Variables

Local Variables

- Local variable: variable that is assigned a value inside a function
  - Belongs to the function in which it was created
    - Only statements inside that function can access it, error will occur if another function tries to access the variable



Local Variables

```
def getName ( ) :
    name = input("Enter your name :")

def main():
    getName()
    print ( name )

# Call the main function.
main()
```

```
Enter your name : Bruno
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/var/folders/9x/hx0451rj4n1fcxsm1m6ch5hx7p3m0h/T/ipykernel_18133/1153231484.py in <module>
      7
      8 # Call the main function.
----> 9 main()

/var/folders/9x/hx0451rj4n1fcxsm1m6ch5hx7p3m0h/T/ipykernel_18133/1153231484.py in main()
      4 def main():
      5     getName()
----> 6     print ( name )
      7
      8 # Call the main function.

NameError: name 'name' is not defined
```

Local Variables

- Different functions may have local variables with the **same name**
- Each function does not see the other function's local variables, so no confusion.

```
def newGreeting():
    name = 'Fred'
    print ('hello', name)
def main():
    name = 'John'
    print ('hello', name)
    newGreeting()
    print ('hello', name)
# Call the main function.
main()
```

```
hello John
hello Fred
hello John
```

Two separate variables called name. One is local to the main and the other is local to newGreeting.

Global Variables

- Global variable: created by assignment statement written **outside all the functions**
  - Can be accessed by any statement in the program file, including from within a function

```
# Create a global variable.
myValue = 10

# The show-value function prints
# the value of the global variable.
def showValue():
    print (myValue)

# Call the show-value function.
showValue()
```

```
10
```

Global Variables

```
# Create a global variable.
myValue = 10

def showValue():
    myValue = 12

def main():
    showValue()
    print (myValue)
# Call the main function.
main()
```

```
10
```

The myValue variable created in the function showValue is a local variable and not the global variable.

Changing a Global Variable Value

```
# Create a global variable.
myValue = 10

def showValue():
    global myValue
    myValue = 12

def main():
    showValue()
    print (myValue)
# Call the main function.
main()
```

```
12
```

The myValue variable created in the function showValue is now a global variable.

If a function needs to assign a value to the global variable, the global variable must be re-declared within the function.

General format: global *variable_name*

Passing Arguments to Functions

- **Argument**: piece of data that is sent into a function
  - When calling the function, the **argument** is placed **in parentheses** following the function name
  - `functionName(argument)`

- **Parameter** variable: variable that is **assigned the value of an argument** when the function is called.
  - The parameter and the argument reference the same value
  - General format:

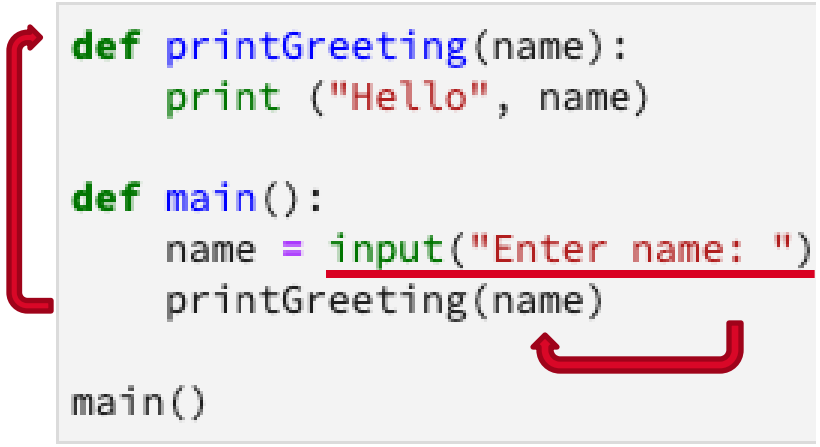
```
def function_name(parameter):
```

  - Scope of a parameter: the function in which the parameter is used

Passing Arguments to Functions

```
def main():
    value = 5
    show_double(value)


def show_double(number):
    result = number * 2
    print(result)
```

Passing Arguments to Functions



```
def printGreeting(name):
    print ("Hello", name)

def main():
    name = input("Enter name: ")
    printGreeting(name)

main()
```

```
Enter name:  Bruno
Hello Bruno
```

Passing Multiple Arguments

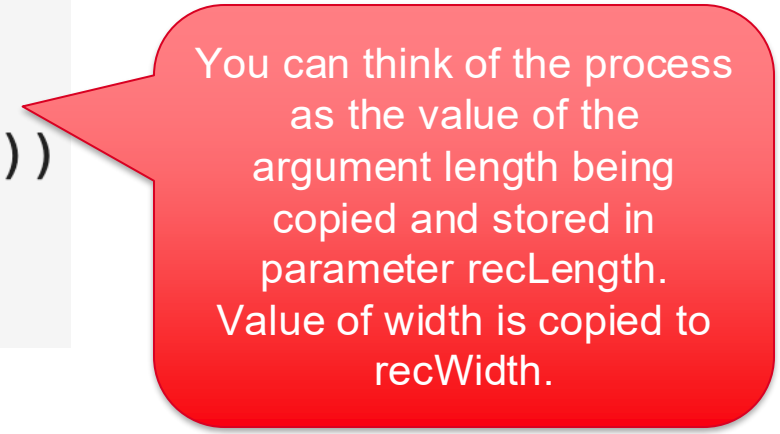
- Python allows writing a function that accepts **multiple arguments**
  - Parameter list replaces single parameter
    - Parameter list items separated by comma

- Arguments are passed ***by position*** to corresponding parameters
  - First parameter receives value of first argument, second parameter receives value of second argument, etc.

- In the example on the next slide we write a function that will calculate the area of a rectangle. It will take in two parameters and print out their product.

Passing Multiple Arguments

```
def calculateArea(recLenght, recWidth):
    print(recLength * recWidth)

def main():
    length = int(input("Enter the length: "))
    width = int(input("Enter the width: "))
    calculateArea(length, width)
```



Passing Multiple Arguments

```
def reverseName(first, last):
    print (last, first)
def main():
    firstName = "Lando"
    lastName = "Calrissian"
    print ("Your name reversed is")
    reverseName(firstName, lastName)

# Call the main function.
main()
```

```
Your name reversed is
Calrissian Lando
```

Passing Arguments to Functions

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Function without inputs

```
def greet_user():
    #Display a simple greeting.
    print("Hello!")

greet_user()
```

```
Hello!
```

Function with one input

```
def greet_user(username):
    """Display a simple greeting."""
    print(f"Hello, {username.title()}!")
greet_user('jesse')
```

```
Hello, Jesse!
```

Function with multiple inputs

```
def SumTwoNumbers(x, y):
    """Sum two numbers"""
    print(x+y)
SumTwoNumbers(2, 3)
```

```
5
```

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Value-Returning Functions

Introduction to Value-Returning Functions:

- **Simple function**: group of statements within a program for performing a specific task
  - Call function when you need to perform the task

- **Value-returning function**: similar to simple function, **returns a value**
  - A value-returning function is a function that returns a value back to the part of the program that called it.
  - The value that is returned from a function can be used like any other value: it can be assigned to a variable, displayed on the screen, used in a mathematical expression (if it is a number), and so on.

Writing Your Own Value-Returning Functions

- We will now look at writing our own value-returning functions in the same way that you write a simple function, with one exception: a value-returning function must have a return statement.
- Here is the general format of a value-returning function definition in Python:

```
#Don't run this cell
def function-name ( ):
    statement
    statement
    etc.
    return expression
```

Writing Value-Returning Methods

- Following is a simple example of a sum method:

```
def sumNum(num1, num2):
    result = num1+num2
    return result
def main():
    total = sumNum(12, 3)
    print ("Sum is", total)

main()
```

```
Sum is 15
```

Returning Boolean Values

- Boolean function: returns either `True` or `False`
  - Normally used to test a condition such as for decision and repetition structures
    - Common calculations, such as whether a number is even, can be easily repeated by calling a function

Boolean Example

```
def isEven(number):
    if (number % 2) == 0:
        status = True
    else:
        status = False
    return status


def main():
    number = 121
    answer = isEven(number)
    print ("Is even is ", answer)

main()
```

```
Is
```

Boolean Example

```
def isEven(number):
    if (number % 2) == 0:
        return True
    else:
        return False


def main():
    number = 122
    answer = isEven(number)
    print ('Is even is ', answer)

main()
```

```
Is even is  True
```

Boolean Example

```
def isEven(number):
    return ( (number % 2) == 0 )


def main():
    number = 122
    answer = isEven(number)
    print ('Is even is ', answer)

main()
```

```
Is even is  True
```

Returning Multiple Values

- In Python, a function can return multiple values
  - Specified after the return statement separated by commas
    - Format: return *expression1, expression2, etc.*

```
def getName():
    first = input( 'Enter your first name:' )
    last = input( 'Enter your last name: ' )
    return first, last


def main():
    firstName, lastName = getName()
    print (f"First Name: {firstName}, Second Name: {lastName}")

main()
```

```
Enter your first name: Bruno
Enter your last name:  Andrade
First Name: Bruno, Second Name: Andrade
```

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Type annotations

Type annotation

- **You can declare variable types as parameters in functions.**
  - Call function when you need to perform the task.

```
def greetings(name: str, age: int) -> str:
    return f"Hello, my name is {name}, and I am {age} years old"

def calculate_area(length: float, width: float) -> float:
    return length * width

def main():
    name = input("Enter your name: ")
    age = input("Enter your age: ")
    print(greetings(name, age))
    length = input("Enter length: ")
    width = input("Enter width: ")
    print(calculate_area(length, width))
main()
```

Type annotation

```python
def process_numbers_inList(numbers: list[int]) -> float:
    return sum(numbers) / len(numbers)

def get_student_grades() -> dict[str, float]:
    return {"Bruno": 95.5, "Haithem": 87.0}
```

```python
class User:
    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

def create_users(users_data: List[Tuple[str, int]]) -> List[User]:
    return [User(name, age) for name, age in users_data]

def authenticate_user(user: User) -> bool:
    return user.age >= 18
```

Type annotation

- Type hints are optional
- Runtime enforcement
- Backwards compatibility
- Better IDE support
- Use mypy to validate types

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Anonymous (Lambda) functions

Lambda functions

- The so-called *anonymous* functions are a specific type of functions supported by Python.
- They are a way of writing functions with a single statement, and the result will be the return value.

```
def square(number):
    return(number**2)

print(square(4))
```

```
16
```

```
squared = lambda x:x**2

squared(4)
```

```
16
```

Lambda functions

- Lambda functions are conveniently useful!
- Several functions for data analysis will require a function as an argument!!

```
def squared_list(numList, fun):
    return([fun(x) for x in numList])

newList = [0,1,2,3,4,5,6,7,8,9,10]

squared_list(newList, lambda x:x**2)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Lambda functions

- Here we have another example, let's say we want to sort a list with animal names by size and then alphabetically:

```
animals = ['Lion', 'Tiger', 'Elephant', 'Giraffe', 'Zebra', 'Panda', 'Kangaroo',
    'Leopard', 'Cheetah', 'Rhino', 'Hippo', 'Wolf', 'Bear', 'Fox', 'Eagle',
    'Crocodile', 'Dolphin', 'Penguin', 'Koala', 'Monkey'
]

animals.sort(key=lambda animal: (len(animal), animal))

animals
```

```
['Fox',
 'Bear',
 'Lion',
 'Wolf',
 'Eagle',
 'Hippo',
 'Koala',
 'Panda',
 'Rhino',
 'Tiger',
 'Zebra',
 'Monkey',
 'Cheetah',
 'Dolphin',
 'Giraffe',
 'Leopard',
 'Penguin',
 'Elephant',
 'Kangaroo',
 'Crocodile']
```

Lambda functions looks complicated, but they really isn’t. They are in fact an important tool that you need to master.

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

How to proper use docstrings

Docstrings are more than just comments

- Docstrings are used to describe information about your class and functions, such as:
  - Expected input
  - Expected output
  - Arguments
  - Constants
  - Errors

Docstrings are more than just comments

Google style	A simple and clear format that is easy to read.
Numpy style	Similar to Google style but with a focus on scientific computing.
Doctest Style	Designed for easy testing of code examples within the docstring.
ReStructuredText	A markup language for formatting.

Example

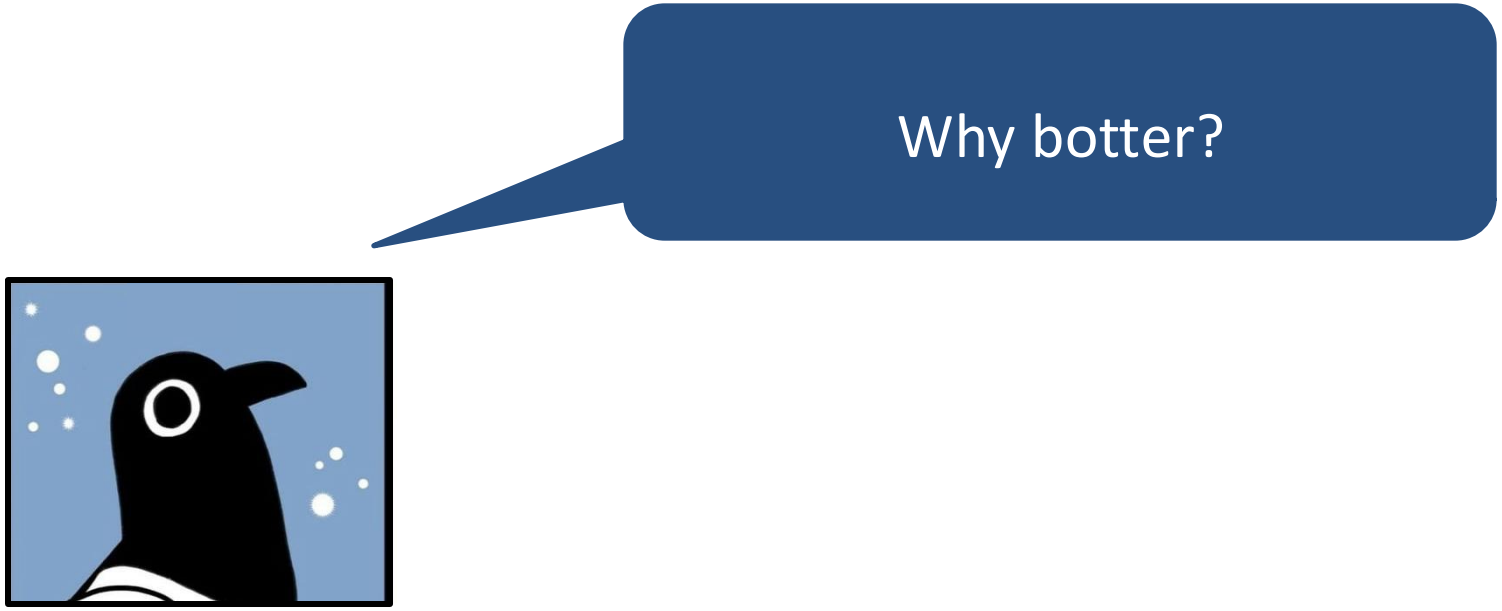
```python
def greetings(name: str, age: int) -> str:
    """Generates a personalized greeting message.

    Args:
        name (str): The person's name to include in the greeting.
        age (int): The person's age to include in the greeting.

    Returns:
        str: A formatted greeting string containing the name and age.

    Example:
        >>> greet("Alice", 25)
        'Hello Alice, you are 25 years old'
        >>> greet("Bob", 30)
        'Hello Bob, you are 30 years old'
    """
    return f"Hello {name}, you are {age} years old"
```

Example

```python
def calculate_area(length: float, width: float) -> float:
    """Calculates the area of a rectangle.

    Args:
        length (float): The length of the rectangle. Must be positive.
        width (float): The width of the rectangle. Must be positive.

    Returns:
        float: The calculated area (length × width).

    Raises:
        ValueError: If either length or width is negative.

    Example:
        >>> calculate_area(5.0, 3.0)
        15.0
        >>> calculate_area(2.5, 4.0)
        10.0
    """
    if length < 0 or width < 0:
        raise ValueError("Length and width must be positive")
    return length * width
```

Example

```python
from typing import List, Tuple

class User:
    """Represents a user with basic profile information.

    Attributes:
        name (str): The user's full name.
        age (int): The user's age in years.
    """

    def __init__(self, name: str, age: int):
        """Initializes a new User instance.

        Args:
            name (str): The user's full name.
            age (int): The user's age in years. Must be non-negative.

        Raises:
            ValueError: If age is negative.
        """
        if age < 0:
            raise ValueError("Age cannot be negative")
        self.name = name
        self.age = age
```


Why botter?

Why botter!

```
Signature: calculate_area(length: float, width: float) -> float
Docstring:
Calculates the area of a rectangle.

Args:
    length (float): The length of the rectangle. Must be positive.
    width (float): The width of the rectangle. Must be positive.

Returns:
    float: The calculated area (length × width).

Raises:
    ValueError: If either length or width is negative.

Example:
    >>> calculate_area(5.0, 3.0)
    15.0
    >>> calculate_area(2.5, 4.0)
    10.0
calculate_area
```

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Analysing code performance

Why optimization matters?

- Optimization is the art of turning a clumsy, memory and computing hungry code (that works!) into something less clumsy, memory and computing hungry code.
- It might involve the use of the right data structures and optimized libraries.

The code should return its output in a timely manner using less resources!

How to do it?

Algorithm	• Avoid nested loops
Data structure	• Different data structures have different trade-offs.
Built-in functions	• Bult-in functions are certainly more optimized than homebrewed ones.
Compiling Python	• Uncommon, but you can compile your Python code by using Cython.
Parallel and distributed computing	• Breaking down the execution of your code through different CPUs will increase it's efficiency.

When to do it?

- When you had to hack something together in a hurry.
- When your Artificial Intelligence model is using more resources than it should, can you improve it somehow?
- If the code needs to return its output for the final user in a satisfactory way.

Beware the premature optimization

- *“Premature optimization is the root of all evil in programming”* - Donald Knuth. *The art of Computer Programming.*

Your code must be working as intended before this step.

Now, how do we decide that the code needs optimization?

By collecting data.

Timing your code

- Simplest and fastest way.
- Only make one change per measurement, please!

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

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

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

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

```
1.63 s
```

So, 1.63s…

Was it good enough?

Timing your code

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

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

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

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

```
1.63 s
```

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

```
0.07 s
```

Timing your code

```
%%timeit
# Creates a list c by adding the elements of a and b
a = list(range(10000000))
b = list(range(10000000))
c = []
for i in range(len(a)):
  c.append(a[i] + b[i])
```

```
489 ms ± 12.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
a = np.arange(10000000, dtype=int)
b = np.arange(10000000, dtype=int)
c = a + b
```

```
20.2 ms ± 419 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Profiling your code

```python
def list_Example():
    a = list(range(10000000))
    b = list(range(10000000))
    c = []

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

```
%%prun
list_Example()
```

```
         10000545 function calls (10000534 primitive calls) in 2.254 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      2/1    0.655    0.327    0.974    0.974 {built-in method time.sleep}
 10000005    0.613    0.000    0.613    0.000 {method 'append' of 'list' objects}
        1    0.566    0.566    0.846    0.846 1184138905.py:1(list_Example)
      5/3    0.212    0.042    0.127    0.042 events.py:86(_run)
        1    0.185    0.185    2.004    2.004 <string>:1(<module>)
      2/1    0.015    0.008    0.012    0.012 {method 'control' of 'select.kqueue' objects}
```

Profiling your code

```python
def numpy_Example():
    a = np.arange(10000000, dtype=int)
    b = np.arange(10000000, dtype=int)
    c = a + b
    return c
```

```
%%prun
numpy_Example()
```

```
         540 function calls (530 primitive calls) in 0.084 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.037    0.037    0.072    0.072 3912121546.py:1(numpy_Example)
        2    0.035    0.018    0.035    0.018 {built-in method numpy.arange}
        1    0.005    0.005    0.077    0.077 <string>:1(<module>)
        2    0.001    0.001    0.001    0.001 {method '__exit__' of 'sqlite3.Connection' objects}
        2    0.001    0.000    0.001    0.001 iostream.py:276(<lambda>)
        1    0.001    0.001    0.001    0.001 {method 'execute' of 'sqlite3.Connection' objects}
```

Profiling your code line by line

```
%lprun -f list_Example list_Example()
```

```
Timer unit: 1e-09 s

Total time: 6.1739 s
File: /tmp/ipykernel_33586/1184138905.py
Function: list_Example at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def list_Example():
     2         1  107400000.0 1.07e+08      1.7      a = list(range(10000000))
     3         1   93966000.0  9.4e+07      1.5      b = list(range(10000000))
     4         1       1000.0   1000.0      0.0      c = []
     5
     6  10000001 2783322000.0    278.3     45.1      for i in range(len(a)):
     7  10000000 3189209000.0    318.9     51.7          c.append(a[i] + b[i])
```

Calculating time and memory usage

```python
# Profile numpy_Example
print("Profiling numpy_Example (vectorized):")
start_time = time.time()
result1 = numpy_Example()
numpy_time = time.time() - start_time
print(f"Time: {numpy_time:.4f} seconds")
print(f"Result type: {type(result1)}")
print(f"Result length: {len(result1)}")
print(f"Memory usage: {result1.nbytes / 1024 / 1024:.2f} MB\n")

# Profile list_Example
print("Profiling list_Example (loop-based):")
start_time = time.time()
result2 = list_Example()
list_time = time.time() - start_time
print(f"Time: {list_time:.4f} seconds")
print(f"Result type: {type(result2)}")
print(f"Result length: {len(result2)}")
print(f"Memory usage: {len(result2) * 24 / 1024 / 1024:.2f} MB\n")
```

```
Profiling numpy_Example (vectorized):
Time: 0.0437 seconds
Result type: <class 'numpy.ndarray'>
Result length: 10000000
Memory usage: 76.29 MB

Profiling list_Example (loop-based):
Time: 1.1791 seconds
Result type: <class 'list'>
Result length: 10000000
Memory usage: 228.88 MB
```

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Opening and saving stuff with Python

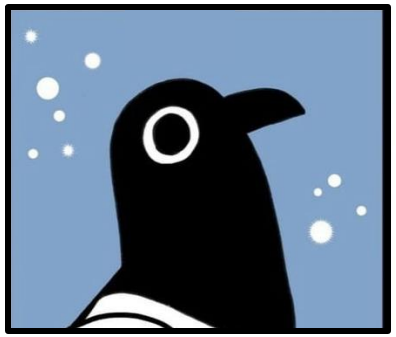
Opening and saving stuff with Python

- Throughout this module we will use high-level tools to read data and to save data into external files.
- But it’s actually quite important to understand the basic of how to works with files in Python.

Opening stuff with Python

- To open a file for reading or writing we use the built-in **open** function.
- To read things from the opened file you use **read**.
- To write things to the opened file (even if it's a new file) you use **write**.
- And to close after you're done, you use **close**.

That is straightforward enough.



Opening stuff with Python

- Ok, let's dive in. We will open a file called haikai.txt

```
path = "Haikai/haikai.txt"

file = open(path, encoding="utf-8")
```

- Here we passed the encoding = "utf-8" as best practice, because the default character encoding can vary from platform to platform.
- Another thing is that these files have permissions, by default we have only reading permission. If we print the file directly we will get the following:

```
file
```

```
<_io.TextIOWrapper name='Haikai/haikai.txt' mode='r' encoding='utf-8'>
```

Opening stuff with Python

- To have access to it's content we need to treat the object like a list and iterate through it:

```
for line in file:
    print(line)
```

```
Wind through autumn leaves

Whispers soft of summer's end,

Moonlit paths ahead.
```

- If you use open to create the file object, you need to close it after you're done to release resources:

```
file.close()
```

Opening stuff with Python

- We can make it easier to do this clean up process by using the “with” statement:

```
haikai2 = "First frost on the grass,\nSilent footsteps trace the dawn,\nCrows call from afar."

path2 = "Haikai/haikai2.txt"

with open(path2, "w", encoding="utf-8") as file:
    file.write(haikai2)
```

```
['Wind through autumn leaves\n',
 "Whispers soft of summer's end,\n",
 'Moonlit paths ahead.']
```

Ok, so file is an object

Does that mean that it has methods?

Methods for the file object

- Yup, of note we have readline, read, seek and tell.
  - readline() – reads one line at a time from the file
  - read() – reads the whole file in one go.
  - tell() – returns the current position of the file cursor.
  - seek() – move the cursor to a specific position in the file.

read()

- With the read method we can bypass the need of a for loop.

```
with open(path, encoding="utf-8") as file:
    haikaiNew = file.read()

haikaiNew
```

```
"Wind through autumn leaves\nWhispers soft of summer's end,\nMoonlit paths ahead."
```

readline()

- With the readline method we can read line by line, one at a time.

```
with open(path, encoding="utf-8") as file:
    line = file.readline()  # Reads the first line
    print(line)

    another_line = file.readline()  # Reads the next line
    print(another_line)
```

```
Wind through autumn leaves

Whispers soft of summer's end,
```

tell()

- Tell will tell you where you are.

```
with open(path, encoding="utf-8") as file:
    print("Current position:", file.tell())  # Output: 0 (at the start)

    file.readline()  # Read the first line
    print("Current position after reading a line:", file.tell())  # Position after reading the first line
```

```
Current position: 0
Current position after reading a line: 27
```

seek()

- seek will move you to another position (byte position!!!)

```python
with open(path, encoding="utf-8") as file:
    print(file.read(10))  # Read the first 10 characters
    print("Current position:", file.tell())  # Current position after reading

    file.seek(0)  # Move cursor back to the beginning
    print(file.read())  # Read the entire file again

    file.seek(10, 0)  # Move the cursor to the 10th byte from the beginning
    print(file.read(10))  # Read the next 10 characters
```

```
Wind throu
Current position: 10
Wind through autumn leaves
Whispers soft of summer's end,
Moonlit paths ahead.
gh autumn
```

What about saving stuff?

read()

- With the read method we can bypass the need of a for loop.

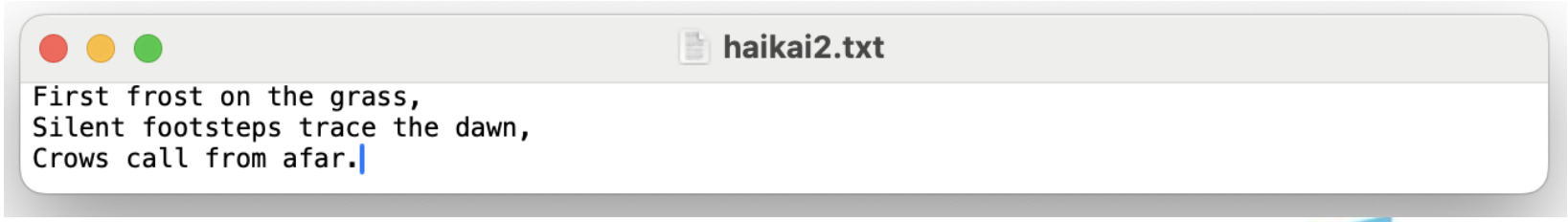
```
with open(path, encoding="utf-8") as file:
    haikaiNew = file.read()

haikaiNew
```

```
"Wind through autumn leaves\nWhispers soft of summer's end,\nMoonlit paths ahead."
```

Saving stuff with Python

- To save stuff with Python is also as straightforward as it gets. To do that let's create a second haikai:

```
haikai2 = "First frost on the grass,\nSilent footsteps trace the dawn,\nCrows call from afar."

path2 = "Haikai/haikai2.txt"

with open(path2, "w", encoding="utf-8") as file:
    file.write(haikai2)
```



Python file modes

Mode	Description
r	Read-only mode
w	Write-only mode; creates a new file (erasing the data for any file with the same name)
x	Write-only mode; creates a new file but fails if the file already exists.
a	Append to existing file; creates the file if it doesn't exist.
r+	Read and write
b	Add to mode for binary files.
t	Text mode for files; this is the default if not specified.

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Exercises

Exercises

1) Write a function that will ask a user for their age and return the value for discount reasons of a show (100 euros per ticket). Write another function that takes in a single int parameter and returns true if the int parameter is greater than 18 or false if it is less than 18. Using the above methods write a program that will ask a user for their age and will inform the user if they are over 18 or not. If the person is over 18 he or she will pay the full price, else, it will be offered a 20% discount! The user should be able to buy more than one ticket, and this option has to be valid for every ticket! Display the final price and ticket details.

2) Write a program that will generate two random numbers between 1 and 100. Write a function that will return the sum of the two numbers. Write a function that will ask the user to guess the value and will return the guessed value. The program should then print out a message indicating if the user has guessed correctly or not.

3) Write a function that will ask the final distance for an extremely drunk person. The function should randomize the steps of the drunk person between -1, 0 and 1. The program has to display how far it went after 10000 steps.

That’s all folks!