PROGRAMACIÓN I

FUNCTIONS

Business Analytics

FUNCTIONS

- *Definition*
- *Arguments and parameters*
- *Annotations/type hinting, docstrings*
- *Execution frames and namespaces*
- *Arbitrary length arguments*
- *Lambda (anonymous functions)*

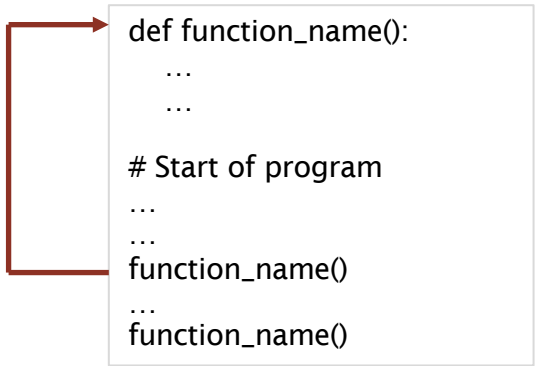
FUNCTIONS

- *Definition*
- *Arguments and parameters*
- *Annotations/type hinting, docstrings*
- *Execution frames and namespaces*
- *Arbitrary length arguments*
- *Lambda (anonymous functions)*

Functions

Universidad Francisco de Vitoria
UFV Madrid

- Groups of related statements
- Parameters, returns a value
- DRY: do not repeat yourself
- When invoked, body of the function is called before control returns.



Built in and user defined functions

Universidad Francisco de Vitoria

UFV Madrid

Built-in functions are those provided by the language and we have seen several of these already:

- For example, both print() and input()

User-defined functions are those written by developers

Function, definition and structure

Universidad Francisco de Vitoria
UFV Madrid

```
def function_name(parameter list):
        """docstring"""
        statement
        statement(s)
```

- All (named) functions are defined using the keyword def
- A function name which uniquely identifies (naming convention lower case _)
- List of parameters (optional): allow data to be passed into the function.
- Colon is used to mark the end of the function header
- Function body, with statements (indented)
- Docstring: documentation string (optional)

Function, definition and structure

Universidad Francisco de Vitoria
UFV Madrid

```
def print_msg():
    print('Hello World!')


print_msg()


def print_my_msg(msg):
    print(msg)


print_my_msg('Hello World')


def square(n: int) -> int:
    return n * n

# Store result from square in a variable
result = square(n=4)
print(result)
```

FUNCTIONS

- *Definition*
- *Arguments and parameters*
- *Annotations/type hinting, docstrings*
- *Execution frames and namespaces*
- *Arbitrary length arguments*
- *Lambda (anonymous functions)*

Arguments and parameters

Universidad Francisco de Vitoria
UFV Madrid

- A parameter is a variable defined as part of the function header and is used to make data available within the function itself.
- An argument is the actual value or data passed into the function when it is called. The data will be held within the parameters.

parameters

```
def make_list_of_range(start: int, end: int) -> list:
    """
    Makes a list of consecutive integers between start and end
    :param start:   first integer
    :param end:     last ingeter
    :return:        a list with consecutive integers
    """

    new_list = list(range(start, end))
    return new_list
```

arguments

```
make_list_of_range(start=2, end=4)
```

Named/keyword arguments

Universidad Francisco de Vitoria
UFV Madrid

- Named arguments (or keyword arguments):

  We provide the name of the parameter we want an argument/value to be assigned to

```
def make_list_of_range(start: int, end: int) -> list:
    """
    Makes a list of consecutive integers between start and end
    :param start:   first integer
    :param end:     last ingeter
    :return:        a list with consecutive integers
    """

    new_list = list(range(start, end))
    return new_list
```

Keyword arguments

```
make_list_of_range(start=2, end=4)
make_list_of_range(end=2, start=4)
```

prog_1/functions/parameters_example.py

Named/keyword arguments

Universidad Francisco de Vitoria
UFV Madrid

- Positional arguments:

  Arguments assigned to parameters based on position

```
def make_list_of_range(start: int, end: int) -> list:
    """
    Makes a list of consecutive integers between start and end
    :param start:   first integer
    :param end:     last ingeter
    :return:        a list with consecutive integers
    """

    newList = list(range(start, end))
    return newList
```

Positional arguments

```
make_list_of_range(2, 4)
```

prog_1/functions/parameters_example.py

Named/keyword arguments

Universidad Francisco de Vitoria
UFV Madrid

- Blend of positional and keyword arguments

```
def make_list_of_range(start: int, end: int) -> list:
    """
    Makes a list of consecutive integers between start and end
    :param start:   first integer
    :param end:     last ingeter
    :return:        a list with consecutive integers
    """

    newList = list(range(start, end))
    return newList
```

Blend of positional and keyword arguments

```
make_list_of_range(2, end=4)
```

prog_1/functions/parameters_example.py

Default values

- With default values:

  Function arguments can have default values in Python.
  We can provide a default value to an argument by using the assignment operator (=).

```
def make_list_of_range_default(start: int, end: int = 10) -> list:
    """
    Makes a list of consecutive integers between start and end
    :param start:   first integer
    :param end:     last integer
    :return:        a list with consecutive integers
    """

    newList = list(range(start, end))
    return newList


make_list_of_range_default(2)
```

prog_1/functions/parameters_example.py

Default values

Universidad Francisco de Vitoria

UFV Madrid

- Any number of arguments in a function can have a default value.
- Once we have a default argument, all the arguments to its right must also have default values.
- That is, non-default arguments cannot follow default arguments.

Return

Universidad Francisco de Vitoria
UFV Madrid

- `return` statement to return a value (or values) to the calling code
- function with terminate
- Return value can be used at the point that the function was invoked

```
def square(n: int) -> int:
    return n * n

# Store result from square in a variable
result = square(4)
print(result)


# Send the result from square immediately to another function
print(square(5))
# Use the result returned form square in a conditional
expression
if square(3) < 15:
    print('Still less than 15')
```

Return, multiple values

Universidad Francisco de Vitoria
UFV Madrid

- It is possible to return multiple values from a function
- We can then assign the values returned to variables at the point when the function is called:

```
def swap(a: int, b: int) -> tuple[int, int]:
    return b, a


a = 2
b = 3
x, y = swap(a, b)
print(x, ',', y)
```

Notice the annotation. May need to import Typing (python < 3.9 or use tuple, P>= 3.9)

FUNCTIONS

- *Definition*
- *Arguments and parameters*
- *Annotations/type hinting, docstrings*
- *Execution frames and namespaces*
- *Arbitrary length arguments*
- *Lambda (anonymous functions)*

Annotations

Universidad Francisco de Vitoria
UFV Madrid

- Python uses dynamic typing
- To provide consistency, we use annotation to “hint” the type of variables
- Particularly useful in parameters and return values of functions
- *No type checking happens at runtime*
- Python assumes separate off-line type checker which users can run over their source code voluntarily
    - This is PyCharm in our case

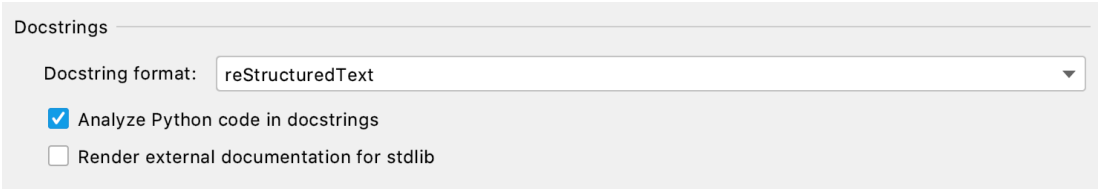
Note: https://www.python.org/dev/peps/pep-0484/#the-meaning-of-annotations

Docstring and annotation (type hinting)

Universidad Francisco de Vitoria
UFV Madrid

```
def make_a_list_annotated(start: int, end: int) -> List:
    """
    Makes a list of consecutive integers between start and end
    :param start:         first integer
    :param end:           last integer
    :return:              a list with consecutive integers
    """

    new_list = list(range(start, end))
    return new_list
```



beginnerspython3/chapter11/helloworldfunction.py

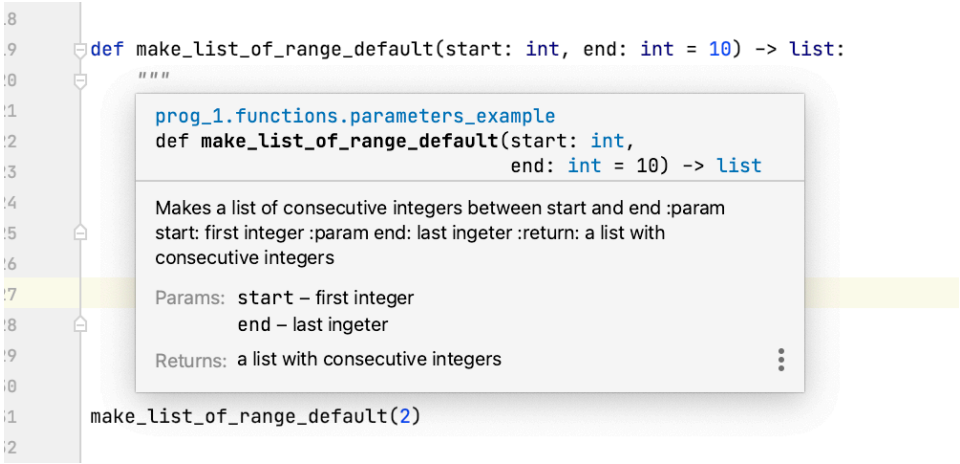
Docstring and annotation (type hinting)

Universidad Francisco de Vitoria
UFV Madrid

```python
def make_a_list_of_range_default(start: int, end: int = 10) -> list:
    """
    Makes a list of consecutive integers between start and end
    :param start:         first integer
    :param end:           last integer
    :return:              a list with consecutive integers
    """

    new_list = list(range(start, end))
    return new_list
```

- Good practice
- IDEs will show documentation when used
- After 6 months, any code, even yours, will look like new
- Required in homework



beginnerspython3/chapter11/helloworldfunction.py

FUNCTIONS

- *Definition*
- *Arguments and parameters*
- *Annotations/type hinting, docstrings*
- *Execution frames and namespaces*
- *Arbitrary length arguments*
- *Lambda (anonymous functions)*

Execution frames, intro

Universidad Francisco de Vitoria
UFV Madrid

- Python creates an execution frame and passes parameters
- Calling code waits for return
- Each execution frame has its own namespace (more on this later)

Calculate a function that returns the nth Fibonacci number

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$

The beginning of the sequence is thus:

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$

Recursive option


```python
def fib_recursive(n: int) -> int:
    """
    Recursive function to calculate the n-th fibonacci number
    :param n:   position in the sequence
    :return:    Fn
    """
    if n == 0:
        # By definition
        return 0
    elif n == 1:
        # By definition
        return 1
    else:
        return fib_recursive(n - 1) + fib_recursive(n - 2)
```

Time Complexity: T(n) = T(n-1) + T(n-2) which is exponential.

Dynamic programming, using arrays


```
def fib_dynamic_array(n: int) -> int:
    """
    Fibonacci Series using Dynamic Programming, using array
    :param n:   position in the sequence
    :return:    Fn
    """
    if n < 2:
        return n
    table = np.zeros(shape=(n + 1,), dtype=int)
    table[1] = 1
    for i in range(2, n + 1):
        table[i] = table[i - 1] + table[i - 2]
    return table[n]
```

Dynamic Programming, using lists


```python
def fib_dynamic_list(n):
    """
    Fibonacci Series using Dynamic Programming, using list
    :param n:   position in the sequence
    :return:    Fn
    """
    if n < 2:
        return n
    results = [0, 1]
    for i in range(2, n + 1):
        results.append(results[i - 1] + results[i - 2])
    return results[-1]
```

Dynamic programming, using iterative approach

Universidad Francisco de Vitoria
UFV Madrid

```
def fib_dynamic_iter(n):
    """
    Fibonacci Series using Dynamic Programming, iterative approach
    :param n:   position in the sequence
    :return:    Fn
    """
    previous, current = (0, 1)
    for i in range(2, n + 1):
        previous, current = (current, previous + current)
    return current
```

Binet's formula


```python
def fib_formula_2(n: int) -> int:
    """
    Fibonacci Series using Binets Formula
    :param n:   position in the sequence
    :return:    Fn
    """
    phi = (1 + 5 ** 0.5) / 2
    return round(phi ** n / 5 ** 0.5)
```

FUNCTIONS

- *Definition*
- *Arguments and parameters*
- *Annotations/type hinting, docstrings*
- *Execution frames and namespaces*
- *Arbitrary length arguments*
- *Lambda (anonymous functions)*

Arbitrary arguments

Universidad Francisco de Vitoria
UFV Madrid

Sometimes, we do not know in advance the number of arguments that will be passed into a function.

- Arbitrary arguments:
  - We can handle this situation through function calls with an arbitrary number of arguments.
  - Syntax: an asterisk (*) before the parameter name to denote this kind of argument.

```
def greeter(*names: str) -> None:
    for name in names:
        print('Welcome', name)


greeter('John', 'Denise', 'Phoebe', 'Adam', 'Gryff', 'Jasmine')
```

```
Welcome John
Welcome Denise
Welcome Phoebe
Welcome Adam
Welcome Gryff
Welcome Jasmine
```

prog_1/functions/parameters_example.py

Arbitrary arguments

Universidad Francisco de Vitoria
UFV Madrid

Use:

- Available as a tuple inside the function, with the name used after the *
- We can use indexes to access individual items (names[0])
- It is iterable: use in a for or as as an argument to another function that takes in an iterable

```
def sorted_greeter(*names: str) -> None:
    print(f'Sorted names: {sorted(names)}')


sorted_greeter('John', 'Denise', 'Phoebe', 'Adam', 'Gryff', 'Jasmine')
```

```
Sorted names: ['Adam', 'Denise', 'Gryff', 'Jasmine', 'John', 'Phoebe']
```

prog_1/functions/parameters_example.py

Arbitrary length keyword arguments

Universidad Francisco de Vitoria
UFV Madrid

It is also possible to pass arbitrary length keyword arguments.

- Syntax ***kwargs* (doble * followed by an identifier)
- *kwargs* becomes a dictionary that maps each keyword to the value that we pass alongside it
- We can iterate using dictionary methods like .item(), .values(), .keys()

```
def values_to_lowercase(**kwargs: str):
    for key, value in kwargs.items():
        print(f"wisper {key} value, {value.lower()}")


values_to_lowercase(first='I', mid='SAY', last='SHOUT')
```

```
wisper first value, i
wisper mid value, say
wisper last value, shout
```

prog_1/functions/parameters_example.py

FUNCTIONS

- *Definition*
- *Arguments and parameters*
- *Annotations/type hinting, docstrings*
- *Execution frames and namespaces*
- *Arbitrary length arguments*
- *Lambda (anonymous functions)*

Anonymous functions lambda

Universidad Francisco de Vitoria
UFV Madrid

When we define a function, we use a name and the object is stored in the namespace

Sometimes we need to build a short, simple function for “immediate” consumption:

- we don’t want to keep it in the name space (to keep in clean and simple, and to avoid names clashes).
- We want to improve readability of the code

Defined using the keyword lambda (lambda functions)

```
def sorting_criteria(student: tuple[str, str, int]) -> int:
    return student[2]
```

becomes, at the point of consumption:

```
lambda student: student[2]
```

prog_1/functions/parameters_example.py

Sorting nested data structures

Universidad Francisco de Vitoria

UFV Madrid

Consider a nested data structure (container):

```
student_tuples = [
    ('john', 'A', 15),
    ('jane', 'B', 12),
    ('dave', 'B', 10),
]
```

We want to sort this structure. After sorting, we will return tuples of str, str, int, the elements of the list.

The question is:

- What criteria are we going to use to sort?

Built in function sorted

sorted(*iterable*, *key=None*, *reverse=False*)

Sorted() has a key parameter to specify a function (or other callable) to be called on each list element prior to making comparisons.

The value of the *key* parameter should be a function (or other callable) that takes a single argument and returns a key to use for sorting purposes.

```
student_tuples = [
    ('john', 'A', 15),
    ('jane', 'B', 12),
    ('dave', 'B', 10),
]
```

Built in function sorted

We are going to sort the tuples based on the integer of the tupple.

Work in groups:

1. Define a function called sorting_criteria that takes in an element of the list and returns the integer
2. Create the variable student_tuples

```
student_tuples = [
    ('john', 'A', 15),
    ('jane', 'B', 12),
    ('dave', 'B', 10),
]
```

3. Call sorted (remember: sorted(*iterable*, *key*, *reverse=False*), and pass the name of the function as argument to the parameter key