Introduction to Programming for Physicists

Dr. Charanjit Kaur

charanjit.kaur@manchester.ac.uk

University of Manchester

PHYS20161 Pre-lecture 2
Semester 1, 2023-24

The material is adapted from previous year's course

Learning Objectives

After covering week 2 material, you will get to know

- Why functions are used in programming and how to write one.
- Scope of variables.
- Logic in programming (and bool data type)
- if statements
- Objects and associated methods

Plan

- Part 1: Functions [Video 1, `function_examples.py`]
- Part 2: Scope [Video 2]

  `add_y_scope_local.py, add_y_scope_global.py, temperature_scope_local.py`
- Part 3: Logic in Programming [Video 3, `bool_example.py`]
- Part 4: `if-elif-else` loops [Video 4]

  `if_example.py, elif_example.py, nested_ifs.py, functions_insteadof_nesting.py`
- Part 5: Objects, attributes & methods [Video 5]

Part 1
Functions

`function_examples.py`

Functions

Mathematical functions

- Functions are mathematical entities

$$y_1 = f_1(x_1)$$

$$y_2 = f_2(x_1, x_2)$$

- We can evaluate these functions as many times as we want for their domain.

Functions in programming (more versatile):

- Could evaluate a (complex) mathematical expression.
- To manipulate a string.
- To write data in a file.
- To print some information
- Etc.

Functions are defined to preform a repeated task.

Built-in Functions

Built-in Functions

A
abs()
aiter()
all()
anext()
any()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

(User-defined) Functions

- Functions are groups of statements executed together to perform some task.
- Defining functions is a more efficient approach to programming:
  - Improve the readability of the code.
  - Defined in one program can be used for the same task in a different program.
  - Improves program adaptability as sometimes changes could be made just by updating functions.

Function Syntax

Defines the following function with a set of input arguments

```
def function_name(input_arguments):
    '''
    Summary of a function
    Input parameters types
    Further details if necessary
    '''


    return return_value
```

Function docstring

Everything indented (4 spaces) is contained within the function.

Specify what to return with return.

If return_value is not explicitly specified or if return statement is not included at all then it will return `None`.

Reserved Keywords

and	as	assert	async	await	break
class	continue	def	del	elif	else
except	finally	for	from	global	if
import	in	is	lambda	nonlocal	not
or	pass	raise	return	try	while
with	yield	False	True	None	

None means no value at all.

Function - Example 1

Functions run in a program when they are called (or invoked) in (after they are defined).

Function name

Keywords

```
def hello_user(user):
    """
    Greets user
    Parameters
    ----------
    user : string

    Returns
    -------
    None

    """
    greeting = 'Hello ' + user
    print(greeting)
    return None
```

docstring

```
hello_user("Name")
```

Called the function

PL2: selections of function_examples.py

Function - Example 2

Functions run in a program when they are called (or invoked) in (after they are defined).

Function name

Keywords

docstring

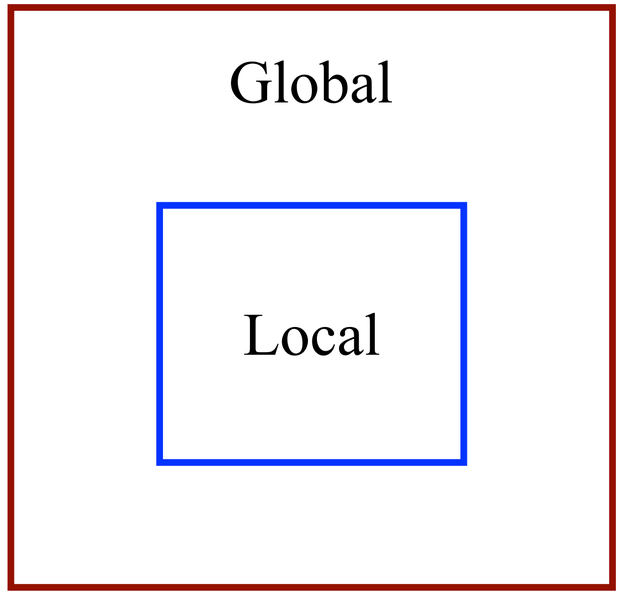
```
def cube_number(number):
    """
    Cube of input number

    Parameters
    ----------
    number : float

    Returns
    -------
    number_cube : float

    """
    number_cube=number**3
    return number_cube
```

```
print(cube_number(5.0))
```

Called the function

PL2: selections of function_examples.py

Part 2
Scope



`add_y_scope_local.py, add_y_scope_global.py,temperature_scope_local.py`

Local Scope Example

```python
def add_y(number):
    """
    Add y to input number

    Parameters
    ----------
    number : float

    Returns
    -------
    result : float

    """
    y = 5.0
    result = number + y
    return result

z=add_y(10.0)
print("z",z)
```

Local Scope of **add_y**

Variables ‘y’ and ‘result’ do not exist outside of this function

PL2: selection of add_y_scope_local.py

Accessing Local Variable

```python
def add_y(number):
    """
    Add y to input number

    Parameters
    ----------
    number : float

    Returns
    -------
    result : float

    """
    y = 5.0
    result = number + y
    return result

z=add_y(10.0)
print("z",z)
print(y)
```

Can't be accessed outside **add_y**

PL2: selection of add_y_scope_local.py

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

Scope: Global Variable

```
Y = 5.

def add_y(number):
    """
    Add y to input number

    Parameters
    ----------
    number : float

    Returns
    -------
    result : float

    """

    result = number + Y
    return result

z=add_y(10.0)
print("z",z)
print("Y",Y)
```

All variables declared here can be used throughout provided they are declared before use.

Variable ‘Y’ exists globally and so can be used by ‘**add_y**’.

Style Note

We use UPPER_CASE for global variables, these should be constant.

We use snake_case for variables within functions.

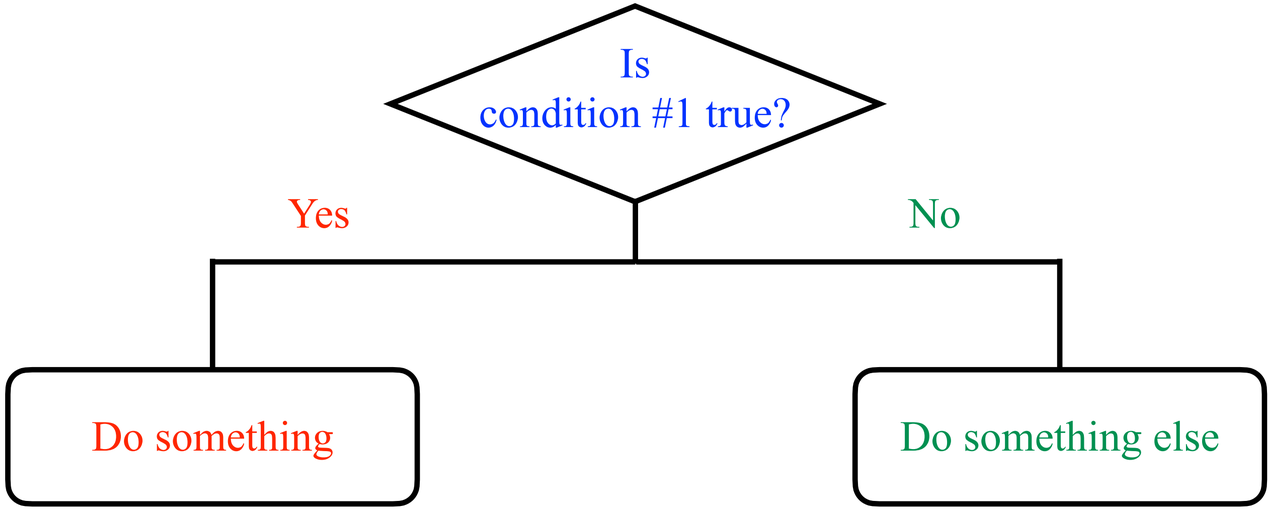
PL2: selection of add_y_scope_global.py

Part 3
Logic in Programming

`bool_example.py`

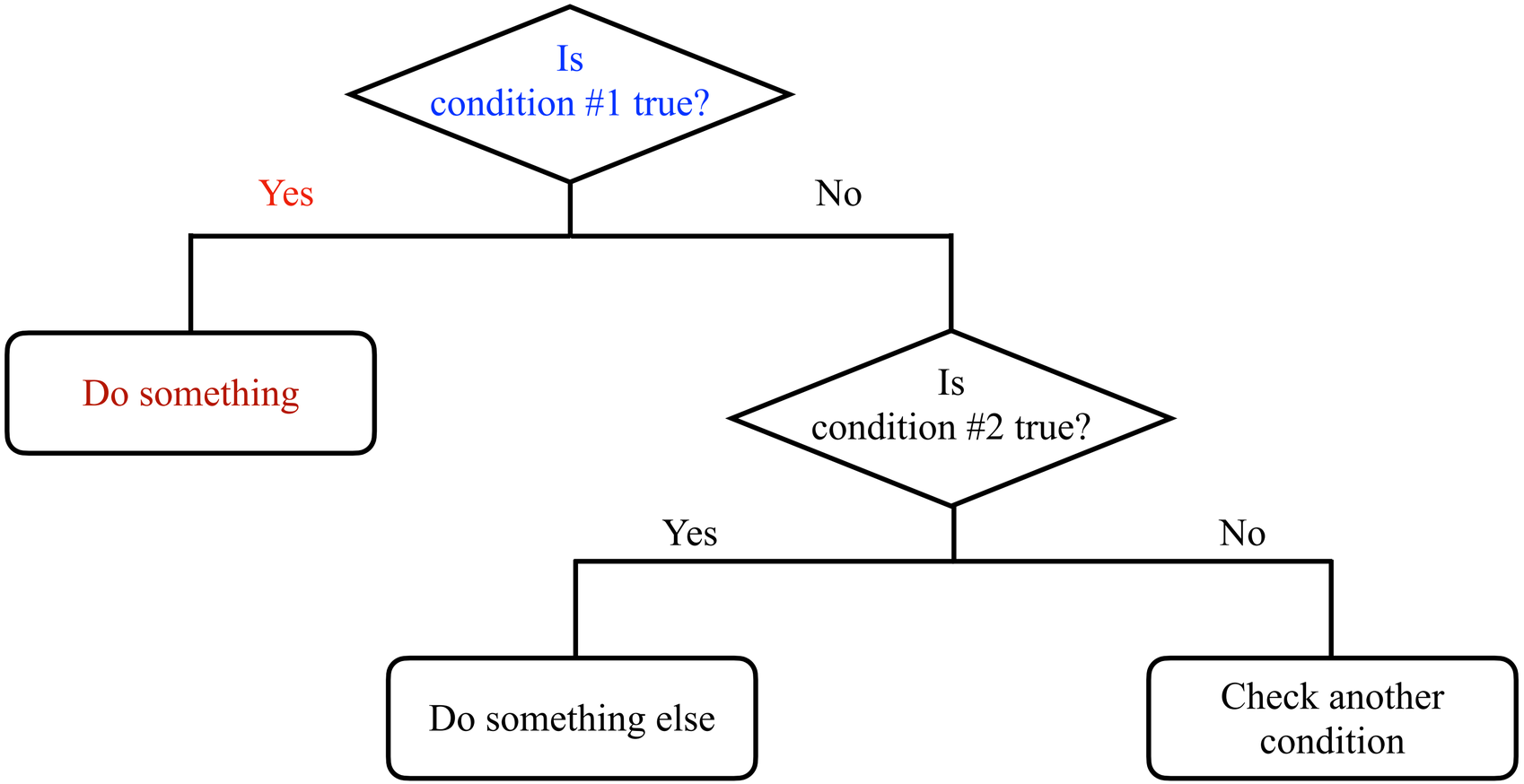
Logical Operations in Programming

Often we want to perform different tasks based on if a condition is true or false.



Logical Operations in Programming

Often we want to perform different tasks based on if a condition is true or false.



Boolean (`bool`) Data Type

A binary variable type:

- True or False
- 1 or 0

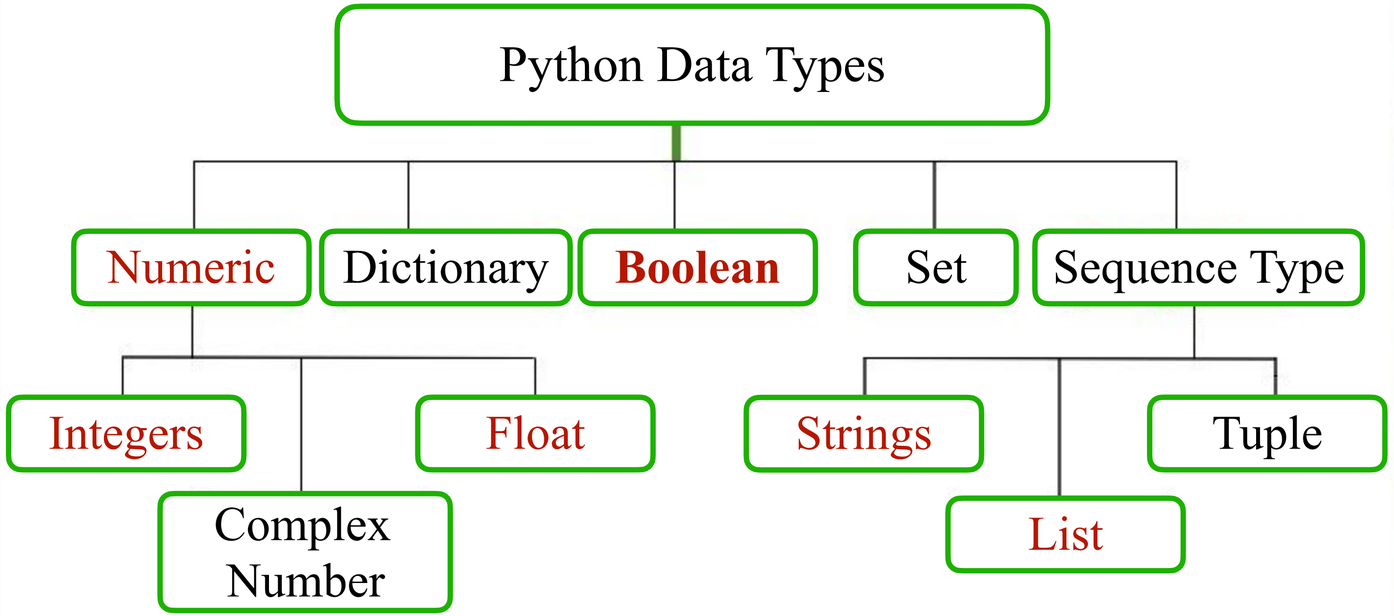


Figure adapted from www.geeksforgeeks.org

Comparison Operators

- == equal to
- != not equal to
- > greater than
- < less than
- >= greater than or equal to
- <= less than or equal to

= *vs* ==

- Assignment Operator (=) assigns value to a variable:
  - E.g. x=4, x is set to equal to 4

- == Operator compares two values then outputs a boolean (true/false)
  - E.g. x==4, is x equal to 4?

Logic Operators

- We can compare multiple aspects using `and`, `or` and `not`.

P	**Q**	**P** or **Q**	**P** and **Q**
True	True	True	True
True	False	True	False
False	True	True	False
False	False	False	False

P	not **P**
True	False
Flase	True

Logic Example

- *Do we have PHYS20161 lecture today?* NO

Answer will be yes if it is a teaching week **and** is Monday.

(teaching week) **and** today = Monday

Is it teaching week? Yes - True

Is it Monday today? No - False

P	Q	P **and** Q
True	True	True
True	False	False
False	True	False
False	False	False

Logic Function Example

```python
def logic_function(bool1, bool2):
    """
    Logic function based on two booleans
    XOR

    Parameters
    ----------
    bool1 : bool
    bool2 : bool

    Returns
    -------
    bool

    """
    return (bool1 or bool2) and not(bool1 and bool2)


if logic_function(True, False):
    print("This is true")


print("Outside of if statement")
```

PL2: selection of bool_example.py

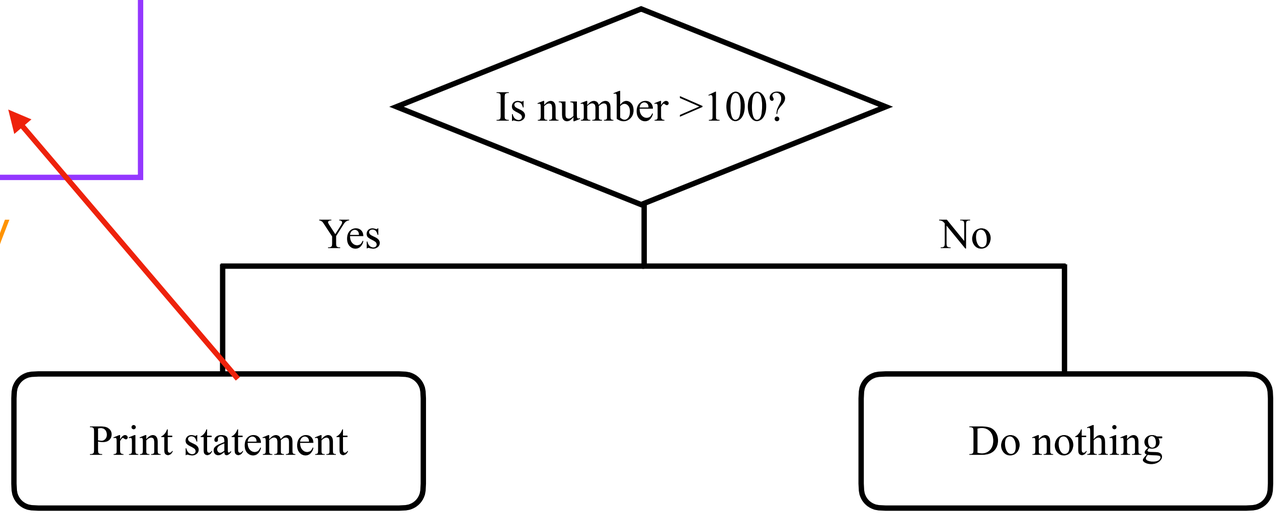
Part 4
if-elif-else

`if_example.py, elif_example.py, nested_ifs.py, functions_insteadof_nesting.py`

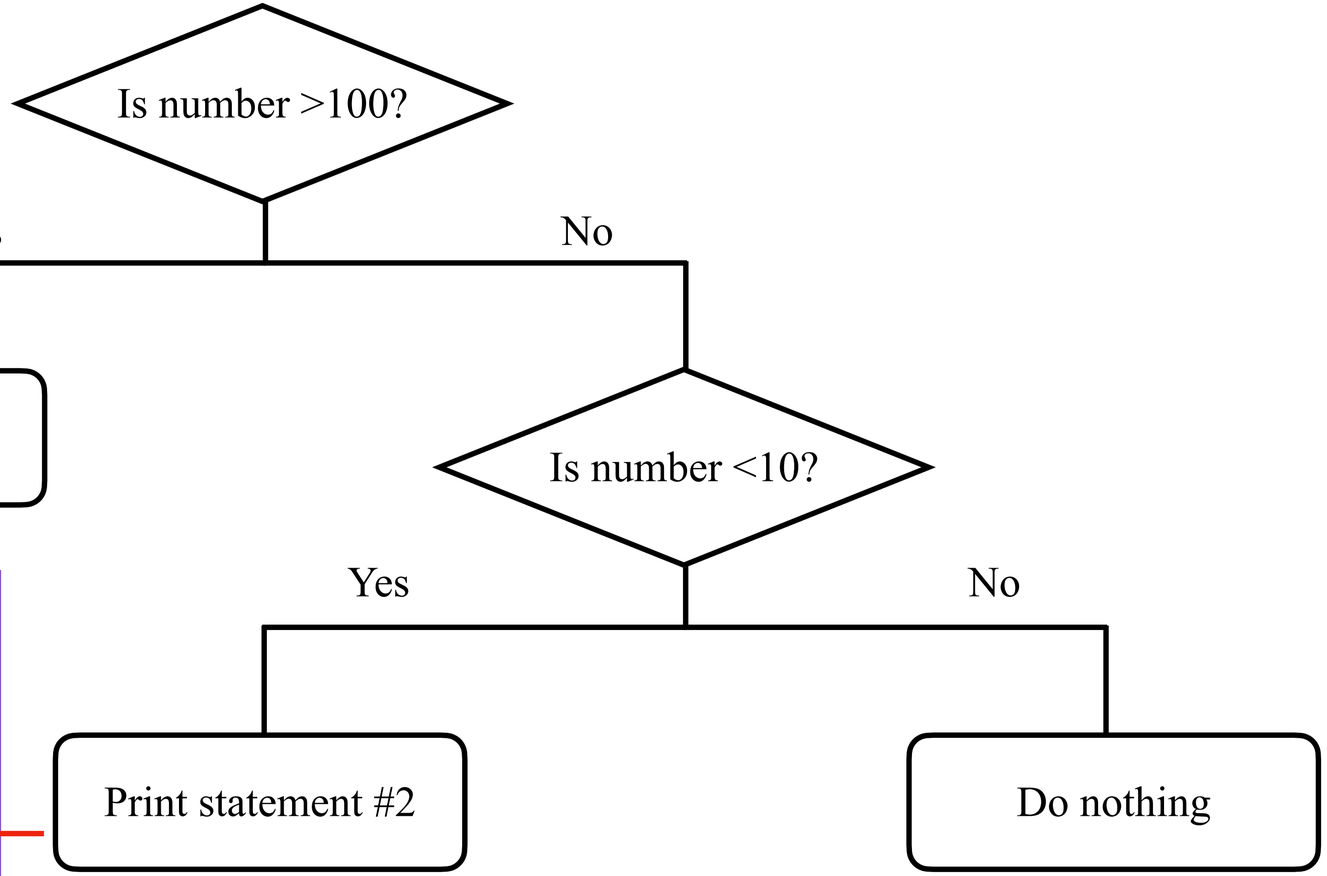
if-elif-else

```
NUMBER = float(input('Enter a number: '))

if NUMBER > 100:
    print("That's greater than 100!")
```

PL2: selection of if_example.py



if-elif-else

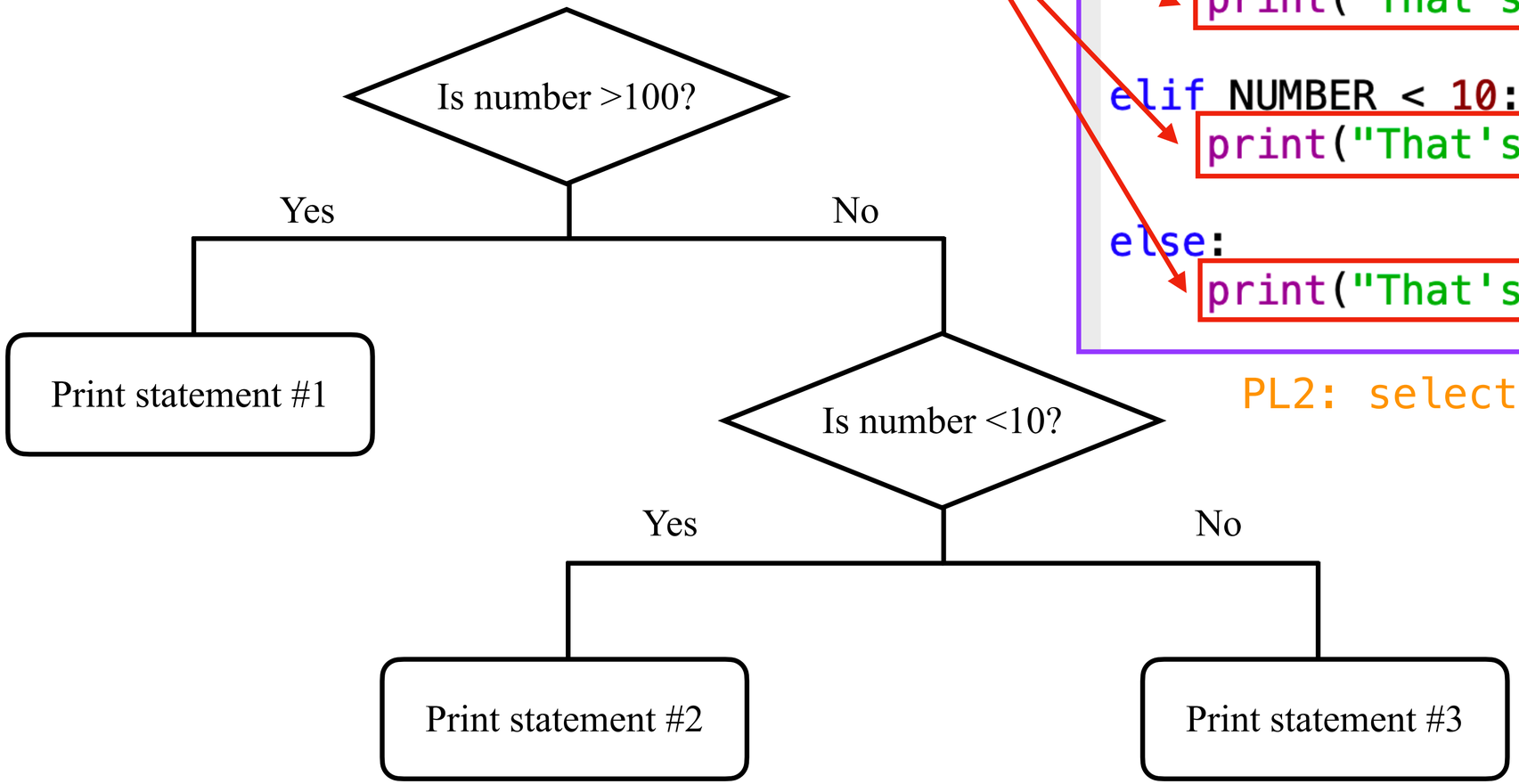



```python
if NUMBER > 100:
    print("That's greater than 100!")

elif NUMBER < 10:
    print("That's less than 10.")
```

PL2: selection of elif_example.py

if-elif-else

Local scope of the if-elif-else statements




```python
if NUMBER > 100:
    print("That's greater than 100!")

elif NUMBER < 10:
    print("That's less than 10.")

else:
    print("That's between 10 and 100.")
```

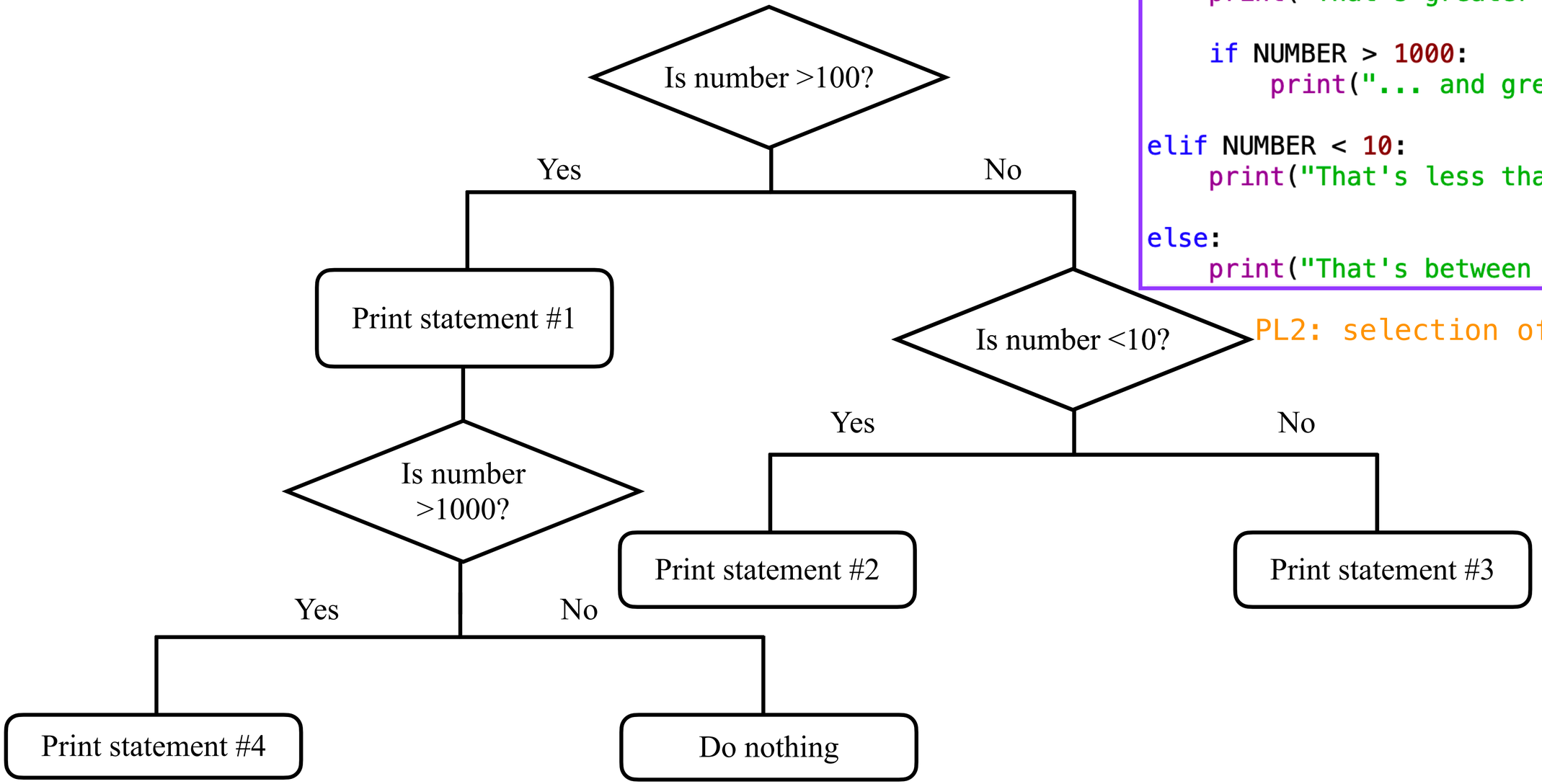
PL2: selection of elif_example.py

Nested if Statement

Nested local scope of the if statement

```
if NUMBER > 100:
    print("That's greater than 100!")

    if NUMBER > 1000:
        print("... and greater than 1000!")

elif NUMBER < 10:
    print("That's less than 10.")

else:
    print("That's between 10 and 100.")
```

PL2: selection of elif_example.py



Nested if-elif-else Statements

- Try to avoid more than 3 levels of indentation (for any statements). Good use of functions allows this, and makes it easier for a reader to understand.

Part 5
Objects

Objects

- Everything in Python is an object that holds data about itself (attributes) and functions (methods) for manipulating the data.
- We can call these attributes/methods with a full stop '.'.

```
In [1]: string = 'Hello'

In [2]: string.lower()
Out[2]: 'hello'
```

- List of methods can be found manually using dir(type).

```
In [3]: dir(float)
Out[3]:
['__abs__',
 '__add__',
 '__bool__',
 '__ceil__',
 '__class__',
 '__delattr__',
 '__dir__',
 '__divmod__',
 '__doc__',
```

Summary

- Functions allow us to define repeated calculations:
  - Improves code readability, adaptability and usability.
- The scope of a variable is set by the indentation of where it is defined:
  - We can define variables locally or globally to suit our needs.
- We can make comparisons between variables to return booleans.
- We can use if-elif-else statements to check a series of comparisons.
- Everything in Python is an object that has callable attributes and methods.

Next week we will cover how to iterate processes with `for` and `while` loops.