Python Programming for Data Science

Week 36, Monday

- Introduction to programming - our first program
- Variables and types:
  - numeric and boolean
  - strings

Introduction to programming - our first program

Programs

A program is a sequence of instructions to a computer

Example: Assembly (low level language)

```
.MODEL SMALL

.CODE

MOV AH,2    ; COMMAND TO PRINT CHAR
MOV DL,'J'  ; CHARACTER TO PRINT
INT 21h     ; INTERRUPT TO PRINT CHAR
INT 20h     ; RETURN SAFELY

END
```

Programs

Higher-level languages are easier to read:

Python

```
for i in range(10):
    print("i = ", i)
```

Java

```
for (int i=0; i<10; i++) {
    System.out.println("i = " + i);
}
```

Ruby

```
for i in 0..9
    puts "i = #{i}"
end
```

C++

```
for (int i=0; i<10; i++) {
    std::cout << "i = " << i << "\n";
}
```

...and they often share concepts and structure.

Compilation or interpretation

Translation of high-level languages into machine code can be done by either:

- compilation
- interpretation
- both

Compilation

Source code is translated from a high-level language into machine code.

It is typically saved to a file for later execution.

Compilation

Source code is translated from a high-level language into machine code.

It is typically saved to a file for later execution.

Results in fastest execution.

Compilation

Source code is translated from a high-level language into machine code.

It is typically saved to a file for later execution.

Results in fastest execution.

Not as common as it used to be - typically used by C, C++ and Fortran.

Interpretation

Source code is translated statement by statement and executed immediately. This is typically what "scripting" refers to.

Interpretation

Source code is translated statement by statement and executed immediately. This is typically what "scripting" refers to.

Results in very slow execution.

Interpretation

Source code is translated statement by statement and executed immediately. This is typically what "scripting" refers to.

Results in very slow execution.

Virtually extinct in modern programming languages. Only shell scripting languages (e.g. bash) do this.

Compilation followed by interpretation

Source code is translated from a high-level language into intermediate language and typically saved to a file.

The intermediate language is interpreted statement by statement. This interpreter is called a Virtual Machine.

Compilation followed by interpretation

Source code is translated from a high-level language into intermediate language and typically saved to a file.

The intermediate language is interpreted statement by statement. This interpreter is called a Virtual Machine.

Execution speed depends on the details.

Compilation followed by interpretation

Source code is translated from a high-level language into intermediate language and typically saved to a file.

The intermediate language is interpreted statement by statement. This interpreter is called a Virtual Machine.

Execution speed depends on the details.

Most modern languages are implemented like this - Java, C#, Perl, Python, Ruby, etc.

Why Python?

Why Python?

Python is a "multi-paradigm" language. It supports procedural, object-oriented and functional programming.

Why Python?

Python is a "multi-paradigm" language. It supports procedural, object-oriented and functional programming.

It has extensive libraries for scientific computing, e.g bioinformatics.

Why Python?

Python is a "multi-paradigm" language. It supports procedural, object-oriented and functional programming.

It has extensive libraries for scientific computing, e.g bioinformatics.

It is (mostly) platform independent.

Why Python?

Python is a "multi-paradigm" language. It supports procedural, object-oriented and functional programming.

It has extensive libraries for scientific computing, e.g bioinformatics.

It is (mostly) platform independent.

It is easy to learn

Why Python?

Python is a "multi-paradigm" language. It supports procedural, object-oriented and functional programming.

It has extensive libraries for scientific computing, e.g bioinformatics.

It is (mostly) platform independent.

It is easy to learn

It encourages (enforces) clean, well-structured code

Python limitations

Speed

Python limitations

Speed

...but this can often be overcome by implementing performance-critical parts in C-modules.

Python2 vs Python3

- Almost the same, but not quite
  - Python3 was released in 2008, as a major cleanup of the Python language. It was not backwards compatible.

Python2 vs Python3

- Almost the same, but not quite
  - Python3 was released in 2008, as a major cleanup of the Python language. It was not backwards compatible.
- Library support?
  - Used to be the main argument for not switching. But basically all libraries are now ported.

Python2 vs Python3

Almost the same, but not quite

Python3 was released in 2008, as a major cleanup of the Python language. It was not backwards compatible.

Library support?

Used to be the main argument for not switching. But basically all libraries are now ported.

"But I don't like Python3"

Get over it. Python2 is dead. Development of Python2 has ceased since 2020.

Python2 vs Python3 (2)

Q: What is the main difference for a beginner like me?

Python2 vs Python3 (2)

Q: What is the main difference for a beginner like me?

A: print behaves differently

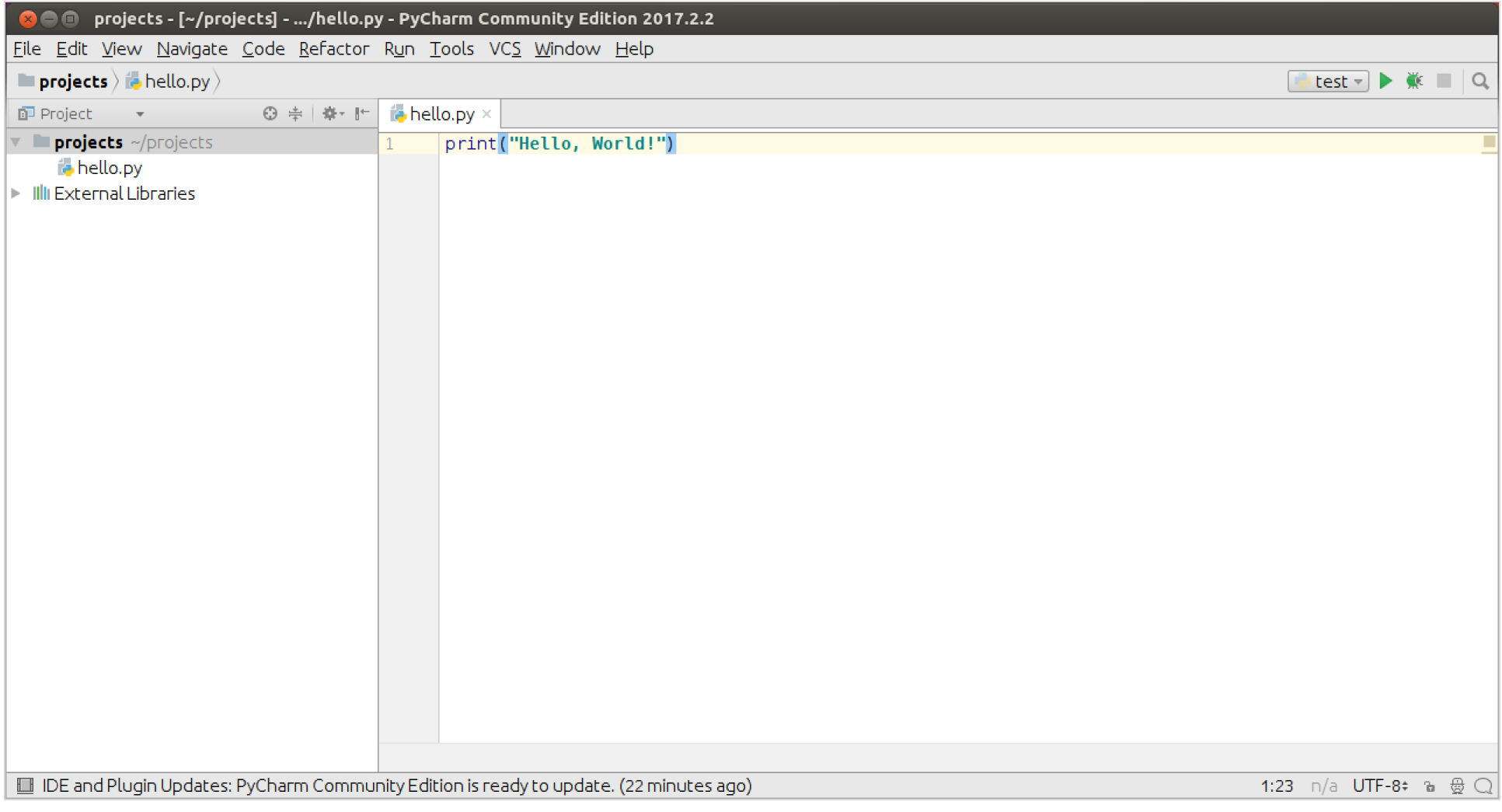
```
print "Hello world!"     # Python 2
print("Hello world!")    # Python 3
```

Jumping in...

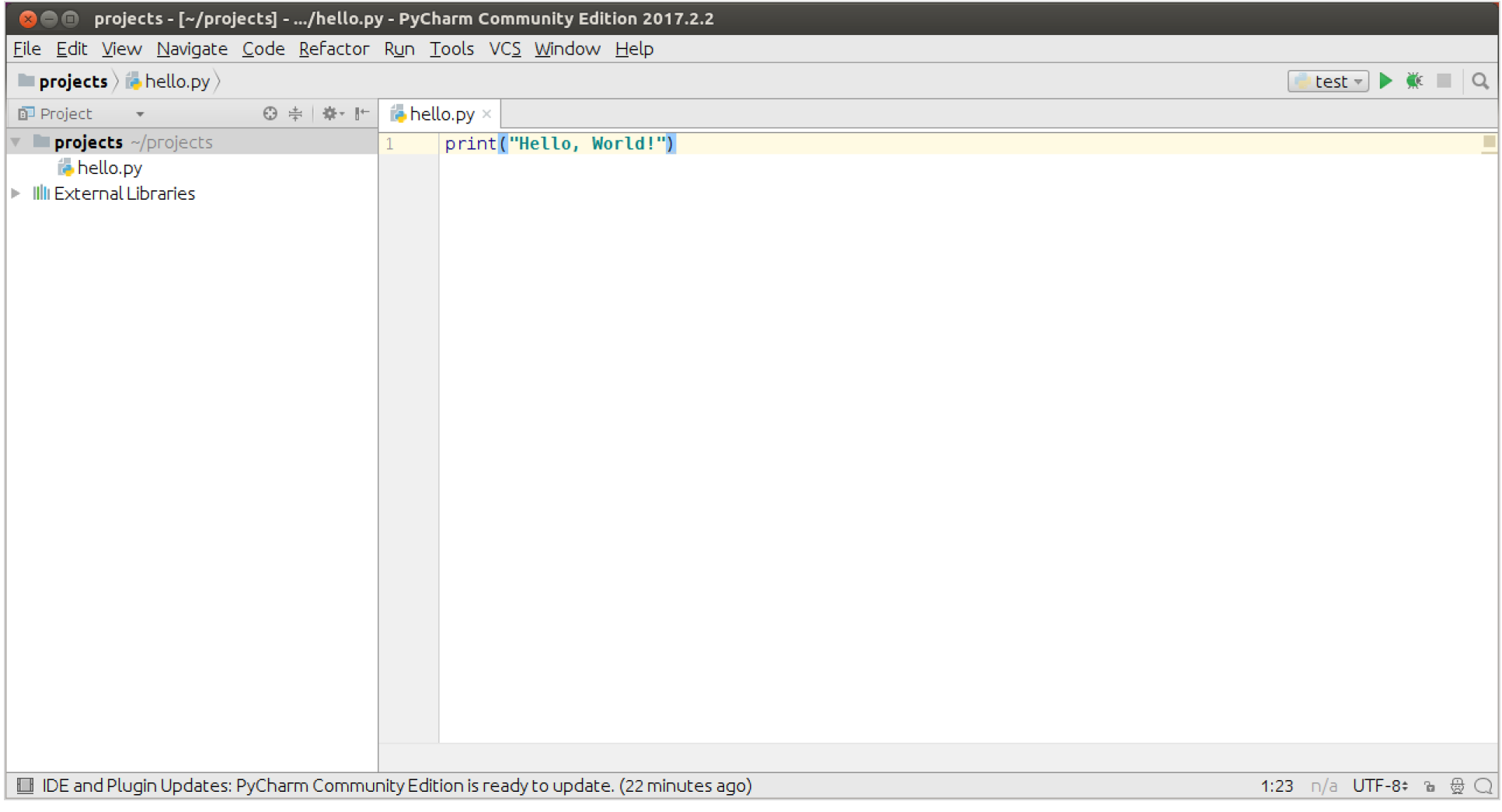
Jumping in...

(If you don't have a working Python installation yet - don't worry: you can do today's basic Python exercises in a browser using Google Colab - click on this link to get started.)

The IDE: PyCharm



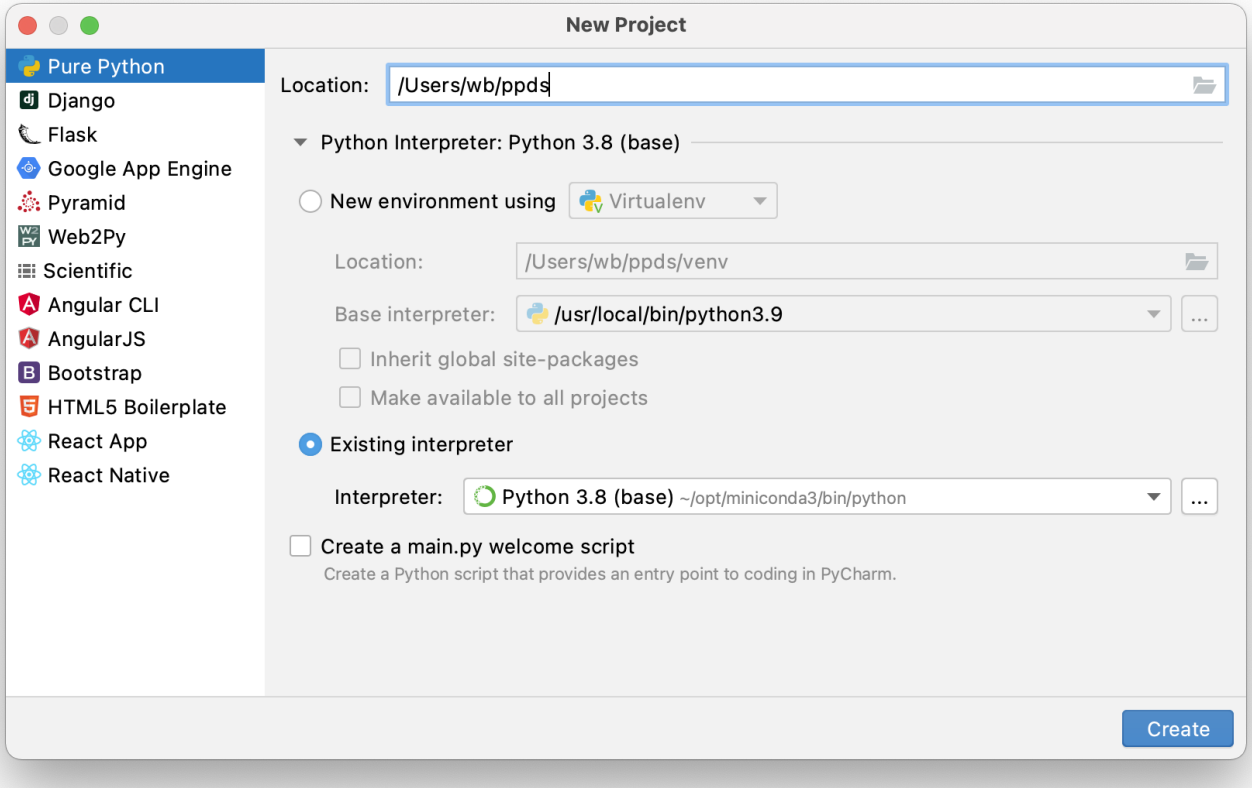
The IDE: PyCharm



The first time it starts, it might take a while to initialize...

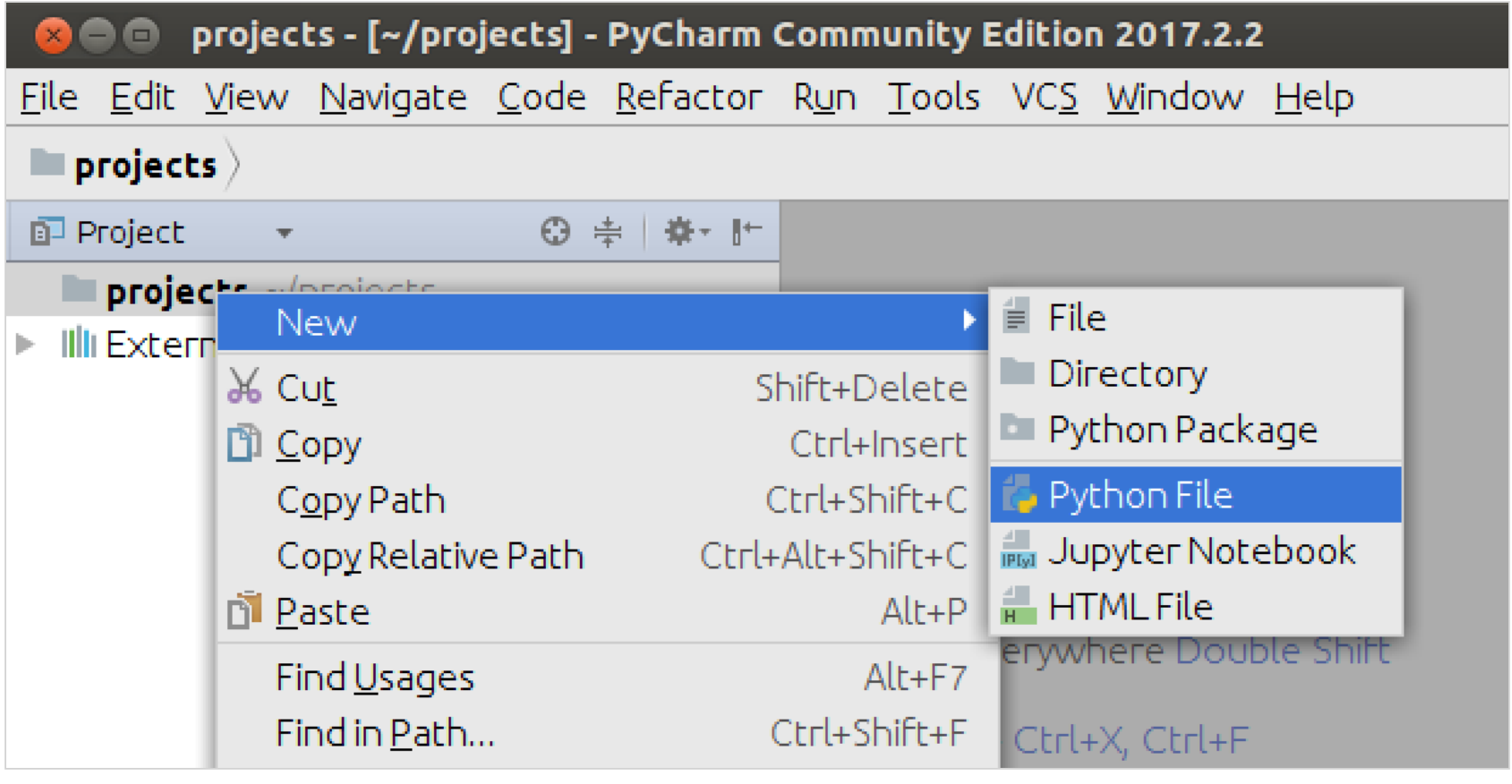
Pycharm: creating a new project

1. Code is organized in projects.
2. When creating a new project, we recommend using the "Existing interpreter" option - and make sure it points to your Anaconda installation



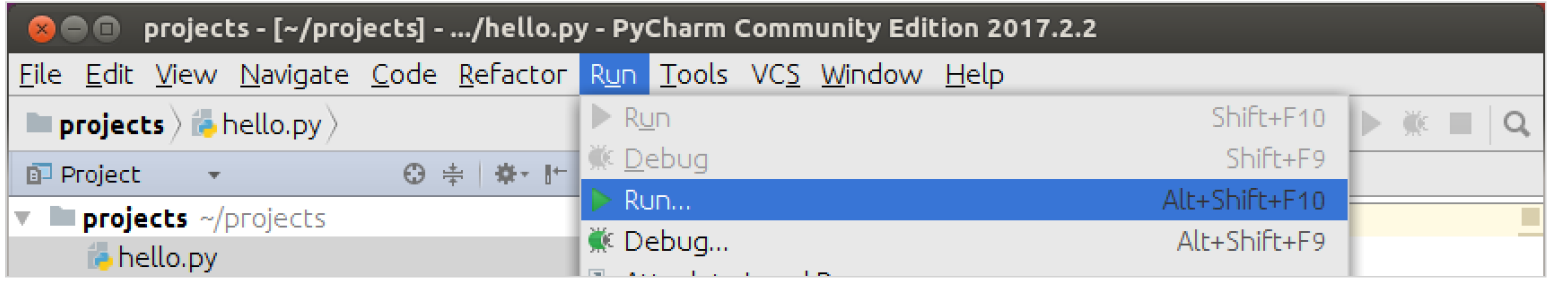
Pycharm: creating a new file

1. Create a new file in the PyCharm IDE.

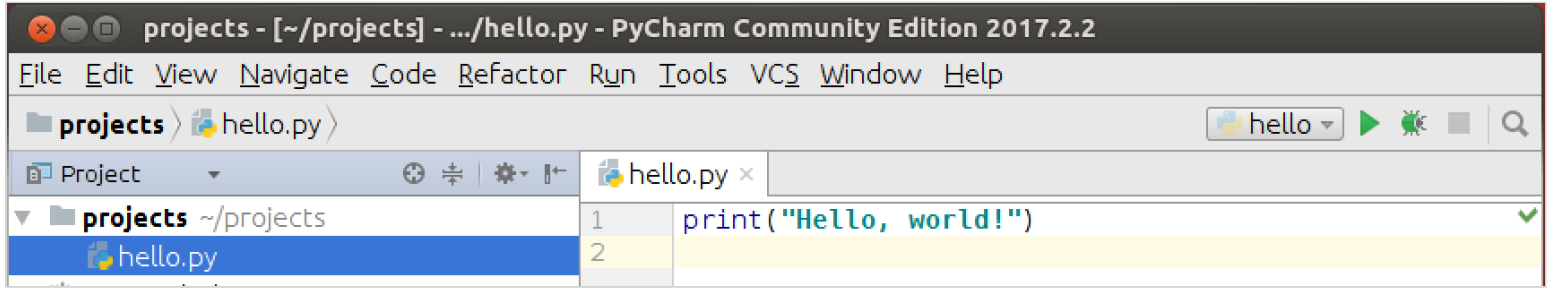


Pycharm: running a script

First time: Select Run from the PyCharm menu



Afterwards: Use the green arrow



Exercise 1: Our first program

1. In PyCharm, create a file called `hello.py`
2. Inside the file, write:

```
print("Hello, World!")
```

3. Run the program twice, once using the menu, and the second time using the green play button

Variables and types: numeric and boolean

Before we start: Python comments

Comments are notes left by the programmer that explain the code.

Comments begin with the # character and continue to the end of the line.

```
# square of the average of all prime numbers between 0 and 10
((2+3+5+7)/4.0)**2
```

Before we start: Python comments

Comments are notes left by the programmer that explain the code.

Comments begin with the # character and continue to the end of the line.

```
# square of the average of all prime numbers between 0 and 10
((2+3+5+7)/4.0)**2
```

In this course, we expect you to hand-in well-documented code. This means that non-trivial lines should have comments.

Variables

Variables are labels that we assign to objects in memory.

Create an integer object and label it as 'a':

```
a = 2
```

Variables

Variables are labels that we assign to objects in memory.

Create an integer object and label it as 'a':

```
a = 2
```

Refer to that memory location:

```
print(a)
print(a * 2)
```

output

```
2
4
```

Variables (2)

In Python, variables do not need to be declared before being assigned a value.

When variables are assigned a new value, the label just starts referring to the new object.

```
a = 2
a = "hello"
print(a)
```

```
hello
```
output

Variables (3)

Name your variables responsibly!

If you are using a well known formula, let's say x = a + b, then you can use 'a' and 'b' as names. Otherwise, be more descriptive:

```
drink_limit = 4
```

Types - Numeric

Three types of numbers:

- Integer - 42
- Float - 42.0
- Complex - 42.0+3.0j

Python arithmetic operators

Python supports the following arithmetic operators:

+	addition	//	integer division
-	subtraction	%	modulo (remainder of division)
*	multiplication	**	exponentiation
/	division	-	negative (unary)

Python2 note: division can be tricky

NOTE: In Python2, the / operator acts as integer division if both operands are integers. If you want normal division, cast one of them as a real number:

```
print(7/2)              # python2: integer division. python3: no problem
print(7/2.0)            # normal division
print(7/float(2))
```

output

```
3    # In python2
3.5
3.5
```

This is NOT a problem in python3

Types - Numeric - comparisons

We can compare two numeric values

```
x = 3
y = 4
print(x < y)
print(x > y)
print(x == y)
```

output

```
True
False
False
```

Types - Numeric - assignment operators

We can perform an operation on a variable and assign the result back to the same variable

The assignment operators are based on the basic arithmetic operators:

```
x = 4
x += 1    # same as x = x+1
x -= 2    # same as x = x-2
x *= 3    # same as x = x*3
x //= 2   # same as x = x//2
x **= 2   # same as x = x**2
x %= 5    # same as x = x%5
```

Types - Boolean

Can take the values of True and False

Operations: 'and', 'or', 'not'

```
x = 1
print((x > 0) and (x < 10))
print((x > 0) or (x < 10))
print((x > 0) and (x > 10))
```

```
True and True    # -> True
True or True     # -> True
```

output

```
True
True
False
```

Types - Boolean

Can take the values of True and False

Operations: 'and', 'or', 'not'

```
x = 1
print((x > 0) and (x < 10))
print((x > 0) or (x < 10))
print((x > 0) and (x > 10))
```

```
True and True    # -> True
True or True     # -> True
```

output

```
True
True
False
```

Note that the two columns are equivalent (for x=1). What would be the right-column equivalent for the third case?

Types - Boolean

Can take the values of True and False

Operations: 'and', 'or', 'not'

```
x = 1
print((x > 0) and (x < 10))
print((x > 0) or (x < 10))
print((x > 0) and (x > 10))
```

```
True and True    # -> True
True or True     # -> True
True and False   # -> False
```

output

```
True
True
False
```

Note that the two columns are equivalent (for x=1). What would be the right-column equivalent for the third case?

Numeric and Boolean types - exercise

1. I want to calculate the average of 4 and 6. What's wrong with the following expression:

```
4+6 / 2
```

2. What is x after executing these lines (try without Python first):

```
x = 4
x += 1
x -= 2
x *= 3
x //= 2
x **= 2
x %= 5
print(x)
```

3. This expression is equivalent to one of the expressions on the previous slide. Which one?

```
(not (not (x > 0) or not (x < 10)))
```

Types - Numeric - Answer

1.
```
(4+6) / 2    # operator precedence: / binds stronger than +.
```

2.
```
x = 4                                  # 4
x += 1
x -= 2
x *= 3
x //= 2
x **= 2
x %= 5
print(x)
```

3.
```
(not (not (x > 0) or not (x < 10)))
(not ((x <= 0) or (x >= 10)))
(x > 0 and x < 10)
```

Types - Numeric - Answer

1.
```
(4+6) / 2    # operator precedence: / binds stronger than +.
```

2.
```
x = 4                                  # 4
x += 1   # same as x = x+1            # 5
x -= 2
x *= 3
x //= 2
x **= 2
x %= 5
print(x)
```

3.
```
(not (not (x > 0) or not (x < 10)))
(not ((x <= 0) or (x >= 10)))
(x > 0 and x < 10)
```

Types - Numeric - Answer

1.
```
(4+6) / 2   # operator precedence: / binds stronger than +.
```

2.
```
x = 4                                   # 4
x += 1  # same as x = x+1              # 5
x -= 2  # same as x = x-2              # 3
x *= 3
x //= 2
x **= 2
x %= 5
print(x)
```

3.
```
(not (not (x > 0) or not (x < 10)))
(not ((x <= 0) or (x >= 10)))
(x > 0 and x < 10)
```

Types - Numeric - Answer

1.
```
(4+6) / 2   # operator precedence: / binds stronger than +.
```

2.
```
x = 4                                   # 4
x += 1  # same as x = x+1              # 5
x -= 2  # same as x = x-2              # 3
x *= 3  # same as x = x*3              # 9
x //= 2
x **= 2
x %= 5
print(x)
```

3.
```
(not (not (x > 0) or not (x < 10)))
(not ((x <= 0) or (x >= 10)))
(x > 0 and x < 10)
```

Types - Numeric - Answer

1.
```
(4+6) / 2   # operator precedence: / binds stronger than +.
```

2.
```
x = 4                                   # 4
x += 1  # same as x = x+1              # 5
x -= 2  # same as x = x-2              # 3
x *= 3  # same as x = x*3              # 9
x //= 2 # same as x = x/2              # 4
x **= 2
x %= 5
print(x)
```

3.
```
(not (not (x > 0) or not (x < 10)))
(not ((x <= 0) or (x >= 10)))
(x > 0 and x < 10)
```

Types - Numeric - Answer

1.
```
(4+6) / 2   # operator precedence: / binds stronger than +.
```

2.
```
x = 4                                   # 4
x += 1  # same as x = x+1              # 5
x -= 2  # same as x = x-2              # 3
x *= 3  # same as x = x*3              # 9
x //= 2 # same as x = x/2              # 4
x **= 2 # same as x = x**2             # 16
x %= 5
print(x)
```

3.
```
(not (not (x > 0) or not (x < 10)))
(not ((x <= 0) or (x >= 10)))
(x > 0 and x < 10)
```

Types - Numeric - Answer

1.
```
(4+6) / 2   # operator precedence: / binds stronger than +.
```

2.
```
x = 4                                   # 4
x += 1  # same as x = x+1              # 5
x -= 2  # same as x = x-2              # 3
x *= 3  # same as x = x*3              # 9
x //= 2 # same as x = x/2              # 4
x **= 2 # same as x = x**2             # 16
x %= 5  # same as x = x%5              # 1
print(x)
```

3.
```
(not (not (x > 0) or not (x < 10)))
(not ((x <= 0) or (x >= 10)))
(x > 0 and x < 10)
```

Strings

Types - Strings

A string is a sequence of characters. It is specified using either:

- single quotes (')
- double quotes(")
- triple quotes(''' or """).

The first two are exactly the same, while the last one is for strings spanning several lines:

```
str1 = "Hello"
str2 = 'Hello'
str3 = '''Hello,
World!'''
```

Types - Strings - operators

Operators for strings work a little differently

+	concatenation
*	repetition
in	test if an element occurs in the string
[]	extract character from string (indexing)

```
str1 = "Linux"
str2 = 'Python'
print(str1 + str2)
print(str1 * 10)
print('ux' in str1)
```

output

```
'LinuxPython'
'LinuxLinuxLinuxLinuxLinuxLinuxLinuxLinuxLinuxLinux'
True
```

Types - Strings - comparisons

Comparisons from strings are done based on how they would appear in a phone book (lexicographically):

>	after
>=	after or equal
<	before
<=	before or equal

```
print("hell" < "hello")
```

```
True
```
output

Types - Strings - special characters

A string can contain combinations of characters that have a special meaning.

They start with the backslash character - \

\n	new line
\t	tab
\'	single quote
\"	double quote
\\	backslash

```
print("1\t2\n3")
```

output

```
1	2
3
```

Converting things to string

Often, you will want to convert values of different types to string — for instance when printing them to screen.

Converting things to string

Often, you will want to convert values of different types to string — for instance when printing them to screen.

This can be done using:

```
str(value)
```

Converting things to string

Often, you will want to convert values of different types to string — for instance when printing them to screen.

This can be done using:

```
str(value)
```

Example:

```
number_of_apples = 6
print("I have " + str(number_of_apples) + " apples.")
```

```
I have 6 apples.
```
output

Types - Strings - exercise 1

1. Create three variables. One called `firstname` with your first name, one called `lastname` with your last name and one called `age` with your age.
2. Merge these strings into one string. When the string is printed to the screen, the output should look like:

```
Name: myfirstname mylastname
Age: age
```

Types - Strings - exercise 1 - solution

- Create three variables. One called firstname with your first name, one called lastname with your last name and one called age with your age.

```
firstname = "Barack"
lastname = "Obama"
age = 60
```

- Merge these strings into one string

Types - Strings - exercise 1 - solution

- Create three variables. One called firstname with your first name, one called lastname with your last name and one called age with your age.

```
firstname = "Barack"
lastname = "Obama"
age = 60
```

- Merge these strings into one string

```
print("Name: " + firstname + " " + lastname + "\nAge: " + str(age))
```

output

```
Name: Barack Obama
Age: 60
```

Types: Strings — indexing

You can index in strings using square brackets:

```
stringname[n]
```

this will return the n+1’th character in the string.

```
s = "hello"
print(s[1])
```

```
’e’
```
output

Note: sequences types in python are zero indexed. The first element has index 0.

Types: Strings — indexing

You can index in strings using square brackets:

```
stringname[n]
```

this will return the n+1'th character in the string.

```
s = "hello"
print(s[1])
```

```
'e'
```
output

Note: sequences types in python are zero indexed. The first element has index 0.

Use a negative index to count from the end of the string:

```
print(s[-1])        # will print 'o'
```

Types are associated with functionality

In an object oriented language, types are typically associated with specific functionality.

Types are associated with functionality

In an object oriented language, types are typically associated with specific functionality.

For instance, for the string-type, it makes sense to have associated functionality for search and replace.

Types are associated with functionality

In an object oriented language, types are typically associated with specific functionality.

For instance, for the string-type, it makes sense to have associated functionality for search and replace.

In order to introduce this associated functionality, we need to very briefly introduce the concepts of *function* and *method*.

Types are associated with functionality

In an object oriented language, types are typically associated with specific functionality.

For instance, for the string-type, it makes sense to have associated functionality for search and replace.

In order to introduce this associated functionality, we need to very briefly introduce the concepts of *function* and *method*.

We'll cover this in much more detail later in the course.

Interlude: functions and methods

A *function* is a piece of code that has a name, and can be *called* from anywhere in the program as:

```
functionname(argument-list)
```

Example:

```
len('hello') # function taking 1 argument
```

```
5
```
output

Interlude: functions and methods

A *function* is a piece of code that has a name, and can be *called* from anywhere in the program as:

```
functionname(argument-list)
```

Example:

```
len('hello') # function taking 1 argument
```

```
5
```
output

A *method* is similar to a function, but it belongs to an *object*. It is called like this:

```
object.methodname(argument-list)
```

```
l = 'hello'
print(l.replace('llo', 'y')) # method taking 2 arguments
```

```
'hey'
```
output

Types: Strings — methods

Commonly used string methods:

`capitalize`	capitalize string
`lstrip`	removing leading whitespace
`center`	center justify
`replace`	replace substring
`count`	count substring occurrences
`rjust`	right justify
`find`	find index of substring
`rstrip`	Remove traling whitespace
`join`	join list of strings
`split`	split into list of smaller strings
`ljust`	left justify
`strip`	`lstrip` and `rstrip`
`lower`	convert to lowercase
`upper`	convert to uppercase

Types: Strings — methods

Examples:

```
s = 'hello'
print(s.upper())
print('hello'.find('ll'))
print('hello'.replace('ll', 'ct'))
print('one two three'.split(' '))
```

output

```
'HELLO'
2
'hecto'
['one', 'two', 'three']
# we'll get back to this last one
```

Types: Strings — exercise 2

1. Create a variable containing a DNA sequence (e.g. AAGCAGAATGCTTAGGACTAGTTAC).
2. Turn it into a RNA sequence ('T' → 'U').
3. Print out the letter in the middle of the sequence.

Types: Strings — exercise 2 — solution

1. Create a variable containing a DNA sequence (e.g. AAGCAGAATGCTTAGGACTAGTTAC).

```
dna_seq = 'AAGCAGAATGCTTAGGACTAGTTAC'
```

2. Turn it into a RNA sequence (’T’ → ’U’).

3. Print out the letter in the middle of the sequence.

Types: Strings — exercise 2 — solution

1. Create a variable containing a DNA sequence (e.g. AAGCAGAATGCTTAGGACTAGTTAC).

```
dna_seq = 'AAGCAGAATGCTTAGGACTAGTTAC'
```

2. Turn it into a RNA sequence (’T’ → ’U’).

```
rna_seq = dna_seq.replace('T', 'U')
print(rna_seq)
```

output

```
AAGCAGAAUGCUUAGGACUAGUUAC
```

3. Print out the letter in the middle of the sequence.

Types: Strings — exercise 2 — solution

1. Create a variable containing a DNA sequence (e.g. AAGCAGAATGCTTAGGACTAGTTAC).

```
dna_seq = 'AAGCAGAATGCTTAGGACTAGTTAC'
```

2. Turn it into a RNA sequence (’T’ → ’U’).

```
rna_seq = dna_seq.replace('T', 'U')
print(rna_seq)
```

```
AAGCAGAAUGCUUAGGACUAGUUAC
```
output

3. Print out the letter in the middle of the sequence.

```
print(rna_seq[len(rna_seq)/2])
```

```
U
```
output