Python Programming for Data Science

Week 36, Monday

Variables and types

- numeric and boolean
- strings

Conditionals (if-else statements)

Introduction to programming

Programs

A program is a sequence of instructions to a computer

Example: Assembly (low level language)

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

END
```

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.
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.
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.
Execution speed depends on the details.

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

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

...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.

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

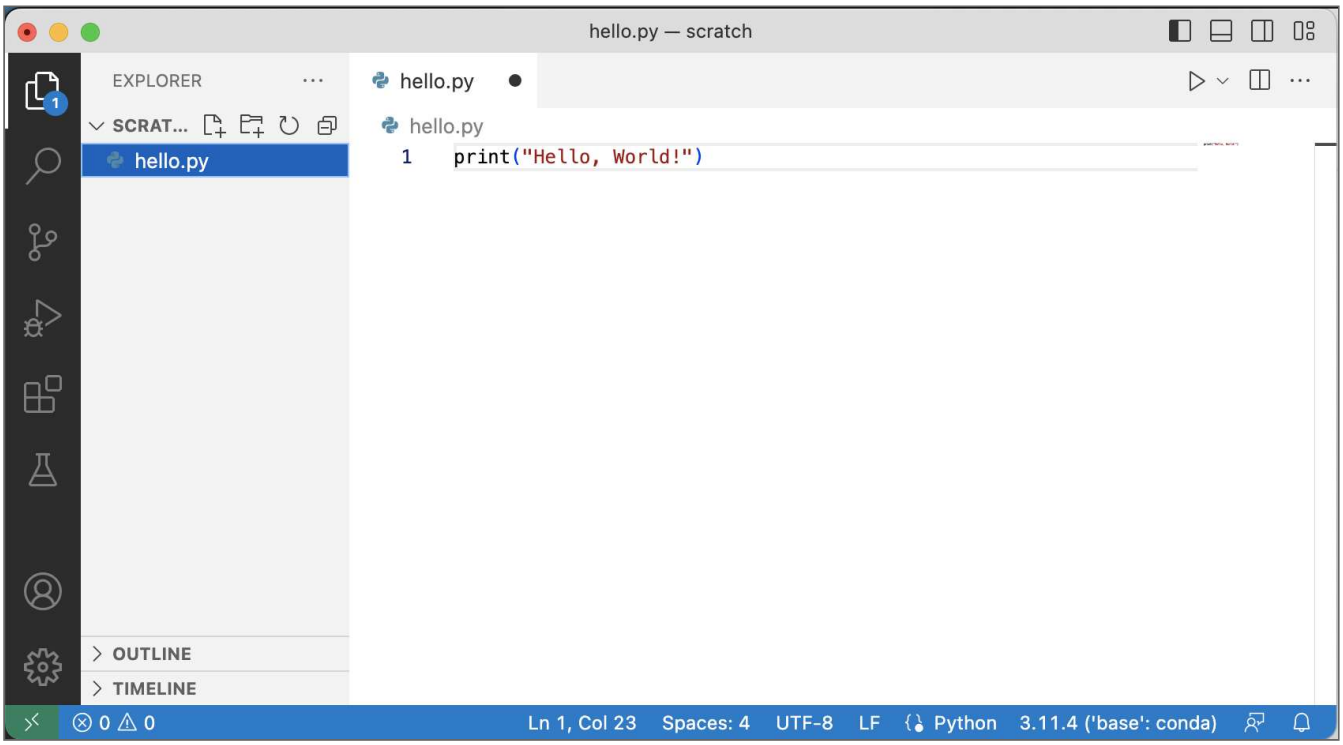
A: print behaves differently

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

Jumping in...

(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: Virtual Studio Code (VS Code)

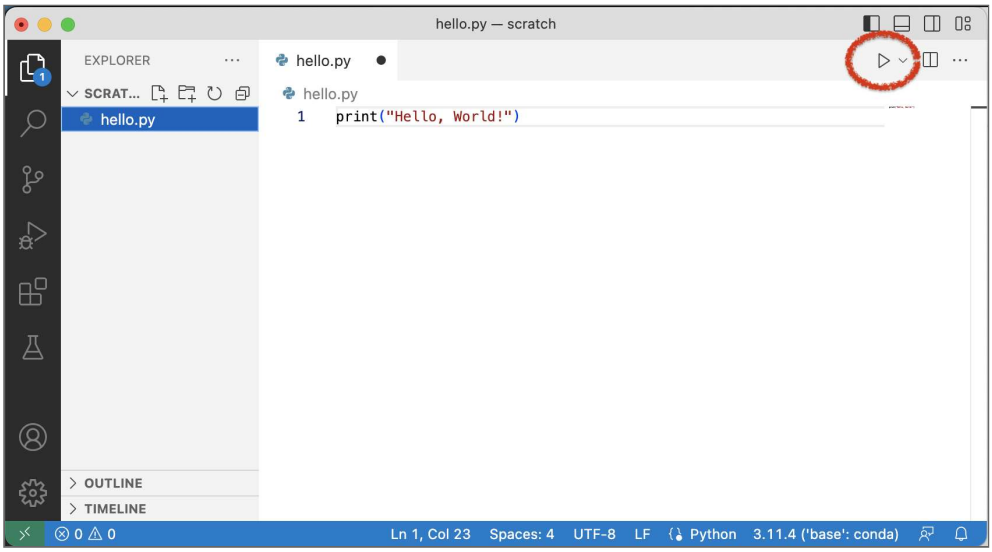


VS Code: opening a folder. opening a file

1. We recommend organizing your code projects in directories, and then choosing `File -> Open Folder` to open such a directory.
2. You can have multiple folders open at once by choosing `File -> New Window`
3. After opening a folder, you can create a file inside this folder by choosing `File -> New File...`

VS Code: running a script

You can run the program you are currently working on by clicking on the play button in the upper right corner



This will open a terminal window, where you can see the output of your program run.

Exercise 1: Our first program

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

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

3. Run the program, and check that it produces the expected output.

Variables and types: numeric and boolean

Before we start: 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
```

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)
```

```
output
hello
```

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
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 - Solution

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 = "Data"
str2 = 'Science'
print(str1 + str2)
print(str1 + " " + str2)
print(str1 * 10)
print('Sc' in str2)
```

```
output
'DataScience'
'Data Science'
'DataDataDataDataDataDataDataDataDataData'
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")
```

```
output
True
```

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

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

```
output
1	2
3
```

\'	single quote
\"	double quote
\\	backslash

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.")
```

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

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 = "code"
age = 62
```

- Merge these strings into one string

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

```
output
Name: Barack Obama
Age: 62
```

Conditionals

(also sometimes called "branching")

Branching - the if statement

The if statement is used to place a condition on a part of your code:

```
if condition:
    code block
```

Example:

```
if 2 < 5:
    print("Two is smaller than five")
```

```
output


Two is smaller than five
```

Note: This is the first (of many) examples where indentation is important to Python.

Branching - boolean expressions

`2 < 5` is a very simple truth expression. More complex expressions can be created using the boolean operators **and**, **or** and **not**:

```
if x >= 30 and x <= 60:
    print("value is within range")

if x < 30 or x > 60:
    print("value is out of range")

if age > 70 and not profession == 'president':
    retired=True
```

Branching - the if-else statement

Often you will want to execute one piece of code when a condition is true, and another if is is false. This is done using the `if-else` statement:

```
if condition:
    code block
else:
    alternative code block
```

```
if 2 < 5:
    print("Two is smaller than five")
else:
    print("Two is larger than five")
```

output

```
Two is smaller than five
```

Note that `if` and `else` must have the same indentation.

Branching - the if-elif-else statement

If you have several mutually exclusive conditions that you want to test for one at a time you can use the general *if-elif-else* version of the if statement:

```
if condition1:
    code block1
elif condition2:
    code block2
elif condition3:
    code block3
.
.
else:
    code block
```

Branching - the if-elif-else statement - example

Simulate a coin toss using a random number generator (RNG). We will import the module **random**, which implements an RNG:

```
import random           # (explained next week)
x = random.random()     # Random number between 0 and 1
if x >= 0 and x < 0.5:      # Is x smaller than 0.5?
    print("Heads")
elif x >= 0.5 and x < 1.0: # Otherwise, is x larger than
0.5?
    print("Tails")
else:
    print("Number not a probability")
```

```
output

Heads
```

Can we simplify this example?

Branching - the if-elif-else statement - example - simpler version

Original

```
import random
x = random.random()
if x >= 0 and x < 0.5:
    print("Heads")
elif x >= 0.5 and x < 1.0:
    print("Tails")
else:
    print("Number not a probability")
```

Simplified

```
import random
x = random.random()     # Random number between 0 and 1
if x < 0.5:             # Is x smaller than 0.5?
    print("Heads")
else:                   # Otherwise, x must be larger than
0.5
    print("Tails")
```

Branches can be nested

```
if condition1:
    if condition2:
        code block2
    elif condition3:
        code block3
    else:
        code block4
```

Example:

```
if x>0
    if y>0:
        print("upper right quadrant")
    else:
        print("lower right quadrant")
else:
    if y>0:
        print("upper left quadrant")
    else:
        print("lower left quadrant")
```

Branching - exercise

1. Use the random number generator from one of the previous slides to generate a random number between 0 and 1, and use this number to simulate throwing a die.

Branching - exercise - solution (1)

1. Use the random number generator from one of the previous slides to generate a random number between 0 and 1, and use this number to simulate throwing a die.

```
import random
# We start by multiplying
with six
# so that our random
number is
# between zero and six
x = random.random() * 6
if x < 1:
    print(1)
elif x < 2:
    print(2)
elif x < 3:
    print(3)
elif x < 4:
    print(4)
elif x < 5:
    print(5)
else:
    print(6)
```

A more clever way to do this:

```
import random
x = random.random()
print(int(x * 6 + 1))
```

Speaker notes