Programming Basics

15-110 – Wednesday 01/14

Announcements

- Over 85% of students completed Ex1-1: well done!
  - If you haven't completed it yet, you still can. Exercises can be submitted late under the revision policy.

- Check1 due next Tuesday at **noon**
  - **Tutorial:** how to download & work on written assignments
  - **Tutorial:** how to submit files on Gradescope

- **No recitation tomorrow**
  - First recitation will be in week 2 when we have material to review

Learning Objectives

- Recognize and use the basic **data types** in programs
- Interpret and react to basic **error messages** caused by programs
- Use **variables** in code and trace the different values they hold

Python and IDEs

Programs are Algorithms for Computers

Computers only know how to do what we tell them to do. **Programs** communicate with a computer and tell it what to do.

Algorithms can be expressed as programs in many different **programming languages**. Different languages use different **syntax** (wording) and commands, but they all share the same set of algorithmic concepts.

In this class, we'll use **Python**, a popular programming language.

An IDE is a Text Editor for Programs

When writing programs, we use IDEs – Integrated Development Environments. These are like text editors for programs.

In this class, we recommend that you use the **Thonny** IDE. It is fairly lightweight, which makes it good for novices.

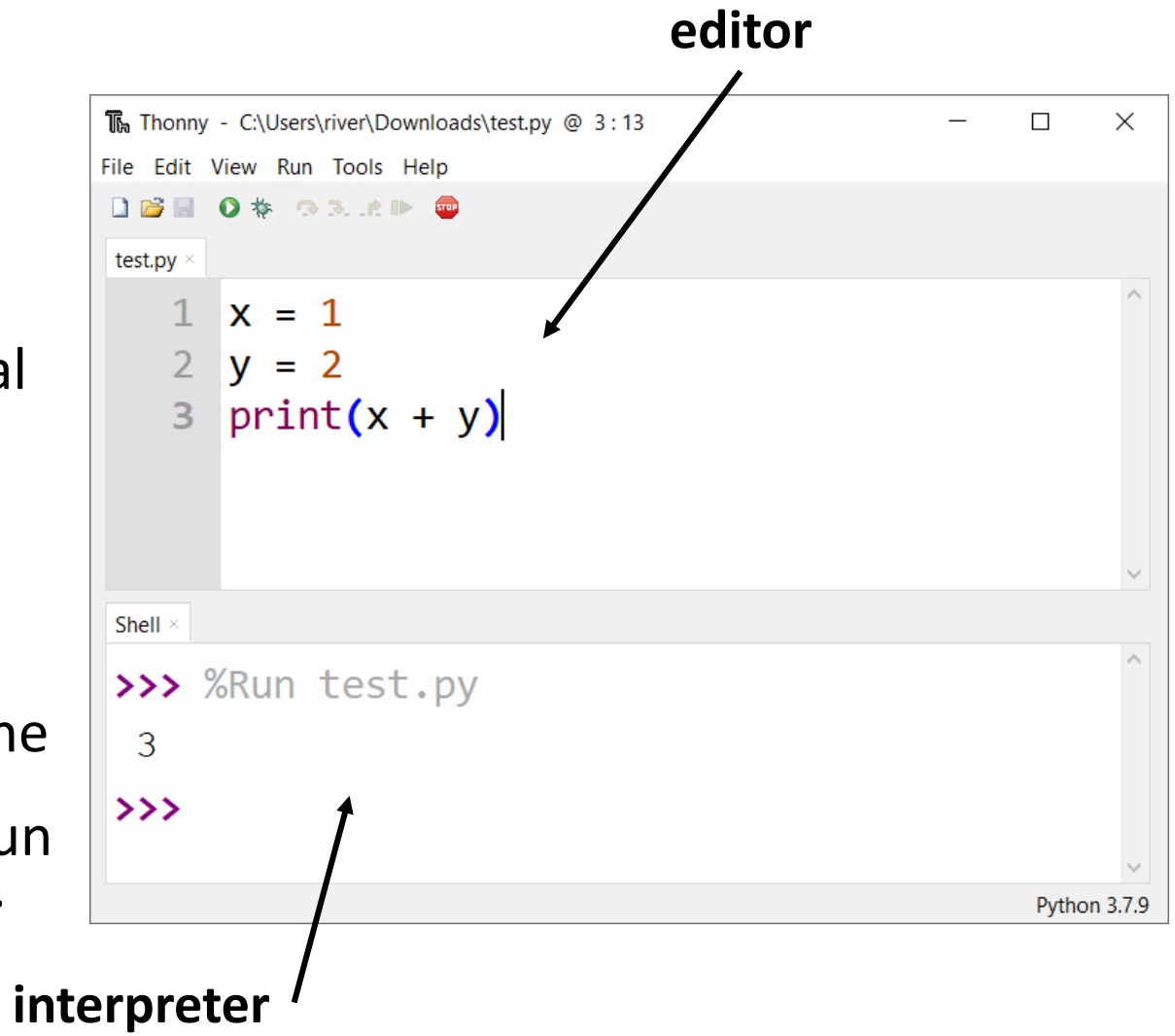
We will mostly use two parts of the Thonny IDE while writing code- the **editor** and the **interpreter**.

Write in the Editor, Run in the Interpreter

The **editor** is just a normal text editor. When we save text, it is saved to a `.py` file, but this is still just normal text.

The **interpreter** (or **shell**) does the actual work of converting our Python text into instructions the computer can run. This happens when you click **Run Current Script** from the **Run** menu.

We can also run single lines of code in the interpreter directly. We'll start by doing that. In general, use the interpreter to run short tasks and the editor for long tasks.



Data Types

Data Is Information We Can Manipulate

Most programs we write will keep track of some kind of information and change it with actions. We call that information **data**.

Data have different **types** depending on their properties. We'll start by going over three core categories: **numbers, text,** and **truth values**.

Data can also be combined using **operations**. We'll show some basic operations for each data type.

Numbers and Operations in Python

Numbers can be represented by two types in Python:

- **Integers** (0, 14, -7) are whole numbers.
- **Floating point numbers** (3.0, 5.735, 8e10) include a decimal point.

Numbers can also be combined using math **operators**:

`+` : addition
`-` : subtraction
`*` : multiplication
`/` : division
`**` : power (2**3 = 8)

Python can combine multiple operations together as a whole and follows order of operations. Use parentheses `( )` to specify the order as needed.

An **expression** like `4**2` or `(5-2)/3` is a piece of code that **evaluates to a data value**. You tell the interpreter to evaluate a piece of code by pressing Enter.

Text in Python

Text values in Python are called **strings**. Text is recognized by Python when it is put inside of quotes, either single quotes (`'Hello'`) or double quotes (`"Hello"`).

Strings can be **concatenated** together using addition.

E.g, `"Hello" + "World"` produces `"HelloWorld"`.

Strings can also be **repeated** using multiplication with an integer!

E.g, `"Hello" * 3` produces `"HelloHelloHello"`

Truth Values in Python

Finally, Python can evaluate whether certain expressions are true or false. These types of values are called **Booleans** after the mathematician George Boole.

Booleans can be either `True` or `False` (no quotes, and capitals are required). These names are built into Python directly.

To get a Boolean, we can write `True` or `False` directly, or do a **comparison**. The basic comparison operators are familiar: `<`, `>`, `<=`, and `>=`.

We can also check if two values are equal (`==`), or not equal (`!=`).

E.g., `"Hello" == "World"` evaluates to `False`

Type Mismatches Cause Errors

Mixing types in Python can cause **error messages** when the types don't go together well. An error message is how the computer tells you it doesn't understand a command you wrote.

For example, "Hello" + 5 results in a TypeError.

```
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Similarly, "Hello" < True results in a TypeError.

On the other hand, integers and floating point numbers **can** be mixed freely. When this happens, the result is usually a floating point number.

For example, 8 * 2.0 results in 16.0

Data Type Names

When reading error messages, note that Python uses shortened names for the four types we've covered.

Integers are called `int`

Floating point numbers are called `float`

Strings are called `str`

Booleans are called `bool`

Activity: Predict the Type

Let's do a poll to see if you can identify data types correctly! For each expression, vote for the type you think it will evaluate to!

Hold up 1, 2, 3, or 4 fingers to indicate your vote:

- 1: **bool**
- 2: **float**
- 3: **int**
- 4: **string**

```
"1" + "2"
```

```
15-110
```

```
"Hello" == "World"
```

```
3.0 * 5.0
```

Writing Code in Files

Writing Longer Programs: Use the Editor

What if we want to run more than one line of code at a time? We'll need to use the **editor**.

Write lines of code in the editor, **save** the file, then click **Run current script**.

Thonny will interpret the entire text file into Python code the computer will understand. It will then run line-by-line through the **entire program** sequentially, where each line is ended by the enter key.

This is different from the interpreter, which ran each line **individually** (though with the context of the previous lines).

Print Displays Data

Code run from a file doesn't show the evaluated result of every line (unlike code run from the interpreter). If we want to display a result, we need to use the command **print**.

`print` takes an input expression between parentheses, evaluates the expression, and displays the evaluated result in the interpreter.

For example, assume we run these lines in the editor:

`print(4 - 2)` displays `2` in the interpreter.

`print("15-110")` displays `15-110`; note that the quotes aren't included.

`5 > 3` does **not** display `True`; it displays nothing, and the result is thrown away.

Printing Multiple Values

If you want to display multiple values in the interpreter on the same line, you have two choices.

First, if you're printing strings, you can concatenate them together.

```
print("Result: " + "2")
```

Alternatively, you can use commas in the `print` command to separate the values. It will then separate the printed values with spaces automatically. This is helpful for printing mixed types.

```
print("Result:", 2)
```

Comments are Ignored by the Computer

When writing a program with multiple lines, you might want to leave notes to yourself or another person outside of the program commands. Use **comments** to do this.

Any text that follows a # on a line will be ignored by the computer:

```
print("Hello World") # a greeting
```

To comment out a block of code, put """ or ''' at the beginning and end:

```
"""
print("ignore")
print("this")
"""
```

You can also select a block of code and click 'Toggle Comment' in Thonny to comment/uncomment a block of code.

Error Messages

Syntax Needs to be Exact

Computers aren't very clever. If you change the syntax of code even a little bit, the computer might not understand what you mean and will raise an error.

```
Print("Hello World") # NameError
print "Hello World" # SyntaxError
```

When you get an error message, **read it carefully**. Error messages contain useful information that will help you fix your code.

Debug Errors By Reading the Message

1. Look for the **line number**. This line tells you approximately where the error occurred.
2. Look at the **error type**.
3. If it says `SyntaxError`, look for the **inline arrow**. The position gives you more information about the location of the problem (though it isn't always right).
4. If it says something else, **read the error message.** The error type and its message give you information about what went wrong.

We'll talk more about the debugging process in future lectures.

1-2.py

```
1 print(Hello World)
2 Print("Hello World")
```

```
>>> %Run test.py
 Traceback (most recent call last):
   File "C:\Users\river\Downloads\test.py", line 1
     print(Hello World)
                     ^
 SyntaxError: invalid syntax

>>>
```

inline arrow

line number

1-2.py

```
1 print("Hello World")
2 Print("Hello World")
```

```
>>> %Run test.py
 Hello World
 Traceback (most recent call last):
   File "C:\Users\river\Downloads\test.py", line 2, in <module>
     Print("Hello World")
 NameError: name 'Print' is not defined

>>>
```

error type

Whitespace is Syntax, Sometimes

Be careful when using whitespace (spaces, tabs, and the return key) – it can sometimes count as syntax too!

In general, whitespace at the **beginning** of a line has meaning; we'll discuss what it means more in a few weeks. Whitespace in the **middle of tokens** causes errors. Whitespace **between tokens** is okay.

```
    print("Hello World") # IndentationError
p r i n t ( "Hello World" ) # SyntaxError
print ( "Hello World" ) # this is okay!
```

Variables

Variables Let Us Store Data

Our last core building block is the **variable**. Variables let us **save data** so we can reuse it in future computations.

A variable is a name that we define in the program (without quotes), like `x` or `result`. We define a variable with an equal sign:

```
variable = value
```

Note that the variable can only go on the left side of this code, and its value (or an expression that evaluates to a value) goes on the right. For example:

```
name = "Sylvi"
result = 5 + 2 # result is set to 7
42 = foo # SyntaxError
```

Variables are like Sticky Notes

You can think of a variable as a sticky note that is applied to a data value.

When you want to use the data value, you can use it directly or refer to the name on the note.

You assign a variable to a value by writing the name on the note and putting the note on the value.

Expressions vs. Statements

Python needs to keep track of certain pieces of data that change over time as a program runs (like which variables exist and what their values are, what has been printed to the screen, etc). We call this information the **program state**.

When you set a variable to a new value, you change the program's state. That makes variable assignments too complex to be represented as expressions (which are more like data values).

A **statement** is an action taken by the program that may change the program state. It does *not* evaluate to a value; instead, it executes a change, then moves on to the next line. Variable assignments are statements.

Variables themselves, on the other hand, actually are expressions – they evaluate to their values!

Using and Updating Variables

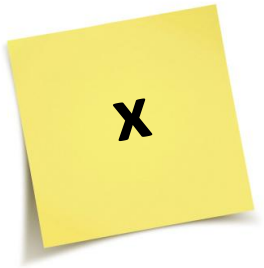
Once we've defined a variable, we can use it in later expressions.

```
x = 5
y = x - 2 # x evaluates to 5
```

Unlike in math, we can also **change** the variable to hold a new value, if needed.

```
x = 5
x = x - 1 # x evaluates to 5 on the right
          # then changes to 4
print("x:", x) # x: 4
```

This is like moving the sticky note to a new value



Python is Sequential

Note that Python runs every line in order and doesn't peek ahead. If you want to use a variable, you must define it **before** it is used.

```
print(foo) # this causes an error!
foo = 42
```

```
foo = 42
print(foo) # this is fine!
```

Activity: Trace the Variable Values

You do: Trace through the following lines of code. What values do a and b hold at the end?

```
a = 4
b = 7
b = a - 2
a = a + 1
```

Sidebar: Rules for Variable Names

Variable names can use any combination of uppercase letters, lowercase letters, digits, and underscores. They must start with a letter or `_`. Starting with a lowercase letter is recommended.

Variable names are case sensitive. For example, `Banana` is not the same as `banana`. Make sure to type your variables correctly, or you'll get a `NameError`!

Learning Objectives

- Recognize and use the basic **data types** in programs
- Interpret and react to basic **error messages** caused by programs
- Use **variables** in code and trace the different values they hold