

Python Data Collection and Management for Public Policy Research

Day 4: Basic Python (Part 1)

Blake Miller[†]

4 July, 2024


[†]Assistant Professor, Department of Methodology, London School of Economics and Political Science (E-mail: b.a.miller@lse.ac.uk)

Agenda for Today

- Debriefing on `git`/Github
- Review of Key Concepts in Programming and Computation
- Basic Python
 - Objects, Types, Variables, Expressions and Statements
 - Functions

Debriefing on `git`/Github

Debriefing on git/Github

- The problem was due to how git handles merge conflicts
- A merge conflict happens when many users simultaneously edit the same lines of code.
- When lines of code are close together, even though they are not on the same line a merge conflict can happen.
- The error had a straightforward solution, but it is beyond what we can cover in this class. See more on merge conflicts [here](#). 
- I've since edited the exercise to minimize the chance of conflicts and suggest you try to complete it yourself outside of class time.

Why are we focusing so much on `git`/Sublime Text/BASH?

- These skills are highly sought after by researchers, admissions committees at social science programs, private sector jobs.
- Managing code, organizing software projects, and problem solving are absolutely essential skills for scientific research.
- By way of analogy: A bicycle is of no use unless you know how to ride it. It is unsafe if you don't have good control of it. It's initially difficult to learn, but becomes automatic after practice.

Plan for Today

- We will need to do some more software installation and setup today, so please everyone choose a partner and sit next to them.
- Try to sit next to someone with the same operating system as you.
- Myself and Yu He will help you if you run into issues. Please raise your hand or approach one of us if you have trouble.

Review of Key Concepts in Programming and Computation

- **Turing complete set:** A minimal set of operations necessary to perform any computation.
- Modern programming languages provide a rich set of **primitives** to simplify coding.
- Example primitives include arithmetic operations, data storage, and control flow mechanisms.
- Programming languages use primitives to create expressions and commands.

Formal and Natural Languages

- Programming languages are formal languages with strict syntax and semantics.
- Unlike natural languages, programming languages are designed to be unambiguous and literal.
- Key terms:
 - **Syntax:** The structure and form of code that a programming language requires.
 - **Static Semantics:** Rules that dictate which syntactically valid strings are meaningful.
 - **Semantics:** The meaning associated with a syntactically correct string of symbols.

Types of Errors

- **Syntactic Errors:** Errors in the code's form or structure.
- **Static Semantic Errors:** the syntax is correct but the operation doesn't make sense semantically.
- **Semantic Errors:** the syntax is correct but it does not do what the programmer intended.

What is a Program?

- A program is a sequence of instructions that specifies how to perform a computation.
- A Python program is a sequence of **definitions** and **commands** evaluated by the **Python interpreter**.
- Python code can be executed in a **shell** or stored in files and run as **scripts**.

Installing Python

Follow instructions on the course GitHub page at
`day_4/01_installing_python.md`

The Python Shell

The Python Interpreter

- An **interpreter** executes **definitions** and **commands** from a script or source code, without compiling the code into machine language first.
- The Python interpreter:
 1. Processes code line by line
 2. Immediately stops execution upon encountering errors

The Python Shell

- The Python shell is an interactive interface for Python programming.
- It allows for quick testing of Python code snippets and direct interaction with the Python interpreter.

Accessing the Python Shell

- Open the command line interface (CLI)
- Type `python` and press Enter.

Example of Using the Python Shell

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


- Python scripts are files with the .py extension.
- A script is usually a series of commands that will be executed sequentially.
- Useful for more complex programs and applications.

Let's Use Script Mode and the Python Shell!

See file on the course GitHub page at `day_4/02_hello_world.py`

Objects, Types, Variables, Expressions and Commands

- Python is an object-oriented programming language, so everything in Python is an **object**.
- Scalar objects represent a single value and cannot be subdivided into smaller pieces (e.g., integers, floats, bools).
- Non-scalar objects can contain multiple elements, can be indexed or iterated over (e.g., strings, lists)
- Every object in Python also has a **type**

Types

- Types define the **operations** that can be done on objects and the values they can hold (e.g., integers, floating-point numbers, strings, and booleans).
- Checking types with the `type()` function:

```
>>> type("Fudan University")
<class 'str'>
>>> type(1)
<class 'int'>
>>> type(True)
<class 'bool'>
>>> type(1.5)
<class 'float'>
```

Type Conversions (Casting)

- Python allows explicit conversion of objects from one type to another.
- Common conversions demonstrated:

```
>>> float(1)
1.0
>>> bool(1)
True
>>> int(1.5)
1
>>> str(1.5)
'1.5'
```

- An **operator** is a symbol or function that indicates an operation to be performed on one or more operands (objects).
- Expressions combine **objects** and **operators** to compute new values.
- Each expression has a value, which has a type.
- A **statement** is a unit of code that has an effect, like creating a variable or displaying a value.

Operators on Ints and Floats

Operation	Operator
Addition	+
Subtraction	-
Multiplication	*
Floor Division	//
Division	/
Modulus (Remainder)	%
Exponentiation	**

Order of Operations

Python follows mathematical conventions for order of operations (PEMDAS):

1. Parentheses
2. Exponentiation
3. Multiplication/Division
4. Addition/Subtraction

```
>>> 2 + 3 * 4 ** 2
50
>>> (2 + 3) * 4 ** 2
80
>>> 2 + (3 * 4) ** 2
146
```

String Operations

- You can concatenate strings using the + operator:

```
>>> first = 'throat'  
>>> second = 'warbler'  
>>> first + second  
'throatwarbler'
```

- Multiplication is useful for repeating strings:

```
>>> 'Spam'*3  
'SpamSpamSpam'
```

Comments

Comments start with “#” and are not executed by the interpreter.

```
# print("This will not print")
>>> print("This will print")
This will print
>>>
>>> # Calculate the percentage of the amount
>>> amount = 120
>>> discount_percentage = 10
>>>
>>> # Calculate discount amount
>>> discount_amount = (discount_percentage /
    100) * amount
>>>
>>> # Print the final price
>>> print(amount - discount_amount)
108.0
```

- **Syntactic Errors:** Errors in the code's form or structure.
- **Static Semantic Errors:** the syntax is correct but the operation doesn't make sense semantically.
- **Semantic Errors:** the syntax is correct but it does not do what the programmer intended.

Types of Errors: Syntactic Error

```
>>> print(Fudan University)
File "<stdin>", line 1
    print(Fudan University)
          ~~~~~~
SyntaxError: invalid syntax. Perhaps you
forgot a comma?
```

“Fudan University” should be enclosed in quotes to be recognized as a string. Without them, the Python interpreter assumes Fudan and University are variables. The `print()` function requires variables to be separated by commas. As the interpreter receives unexpected input, it flags this as invalid syntax.

Types of Errors: Static Semantic Error

```
>>> "number: " + 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "
    int") to str
```

The syntax here is correct, but the "+" operator means something different for strings than it does for integers and floats. For strings, it is used for concatenation, not addition, so this statement does not make semantic sense.

Types of Errors: Semantic Error

```
>>> "1" + "1"  
'11'
```

The syntax and semantics are correct, but the outcome is not what the programmer intended (expecting a numeric addition in string format ('2')) rather than string concatenation). This error happens because both operands are strings, so the + operator performs concatenation.

Variables and Assignment

- A **variable** is a name associated with a location in memory that stores value(s).
- The “=” operator assigns a value (right hand side) to a variable name (left hand side):

```
>>> message = 'And now for something  
              completely different'  
>>> n = 17  
>>> pi = 3.1415926535897931
```

- Naming rules and conventions:
 - Can include letters, numbers, and underscores (_).
 - Must start with a letter or underscore.
 - Cannot include reserved Python keywords (e.g., type, if, for).

Why Variables?

```
>>> pi = 3.14
>>> radius = 10
>>> circumference = pi * (2*radius)
>>> area = pi * (radius**2)
```

- We only need a single update the variable `radius` if we want to now calculate the circumference and area of a circle with a radius of 20.
- We would otherwise need to make two changes to the code if we had not stored `radius` as a variable.
- Variables can be **reassigned** to new values.

Reassignment Example

Note: The value for area does not change until you tell the computer to do the calculation again.

```
>>> pi = 3.14
>>> radius = 2.2
>>> area = pi * (radius ** 2)
>>> print(area)
15.197600000000003
>>> radius = radius + 1 # radius is now 3.2
>>> print(area)
15.197600000000003
>>> area = pi * (radius ** 2)
>>> print(area)
32.153600000000004
```

Floor Division and Modulus

- Floor division (//) divides two numbers and rounds down to an integer.
- Modulus (%) returns the remainder from the division.

```
>>> minutes = 105
>>> hours = minutes // 60
>>> remainder = minutes - hours * 60
>>> print(hours, "hour", remainder, "minutes")
1 hour 45 minutes
```

Functions

What is a Function?

- A function in programming is a reusable block of code designed to perform a specific task.
- Functions are not run until they are called or invoked in a program.
- Functions can have parameters and a return type.

```
>>> def is_even(i):  
...     return i % 2 == 0  
...  
>>> is_even(3) # Calls the function with 3  
           as an argument  
False  
>>>
```

Another Function Example

```
>>> def minutes_to_hours(minutes):  
...     hours = minutes // 60  
...     remainder = minutes - hours * 60  
...     print(hours, "hour", remainder, "  
...         minutes")  
...     return(hours, remainder)  
...  
>>> hours, minutes = minutes_to_hours(105)  
1 hour 45 minutes  
>>> print(hours)  
1  
>>> print(minutes)  
45  
>>> hours, minutes = minutes_to_hours(268)  
4 hour 28 minutes
```

- You may notice that functions include a block of text that is indented.
- Python uses whitespace (spaces and tabs) to define the scope of functions (and also loops, conditionals, and classes, as we will learn later).
- Rules for Indentation:
 - All commands within the block must be indented by the same amount of whitespace.
 - No mixing of tabs and spaces.

Why Functions?

- Functions offer the benefit of **abstraction**: tasks can be defined at a higher level rather than in procedural steps.
- Functions help break a program into smaller, manageable parts, improving **readability** and making a program **easier to debug**.
- Allow for **reuse of code** across different parts of a program or different programs.
- Recall DRY code (**do not repeat yourself**)
 - Any task you run more than once should be a function

Defining Functions

- You can define your own functions to perform specific tasks.
- A function is defined with “def” and is subsequently called using the name and parameters (in parentheses):

```
>>> def say_hello(name):  
...     print("Hello", name)  
...  
>>> say_hello("Dr. Miller")  
Hello Dr. Miller
```

Defining Functions

- Functions can have optional parameters with default values assigned using “=.”
- We do not have to provide these arguments in the function call.

```
>>> def say_hello(name, question="", how are
      you?):
...     print("Hello", name, question)
...
>>> say_hello("Dr. Miller")
Hello Dr. Miller, how are you?
>>> say_hello("Dr. Miller", question="", you
      alright?)
Hello Dr. Miller, you alright?
```

Parameters and Arguments

```
>>> def print_twice(bruce):  
...     print(bruce)  
...     print(bruce)  
...  
>>> print_twice('Spam')  
Spam  
Spam
```

- Parameters are variables used in function definitions (bruce).
- Arguments are values provided during function calls ('Spam').

Understanding Variable Scope

- **Scope** defines the region in a program where a variable is accessible. Each function call creates a new scope or environment where variables are local and mapped to their values.
- **Local vs. Global Scope:** Variables defined within a function are local and not accessible outside it, contrasting with global variables that are accessible across the entire program.
- **Parameter Binding:** Upon a function call, formal parameters (defined in the function) are bound to the actual parameters (passed to the function), establishing their values within the function's scope.

Practical Example: Function Scope

```
>>> def f(x):  
...     x = x + 1  # Local modification of x  
...     print(x)  # Displays the local value  
...     of x  
...     return x  # Returns the modified x  
...  
>>> x = 3  # Global x  
>>> z = f(x)  # Calls f with global x,  
           returns new value to z
```

4

Practical Example: Function Scope (details)

- The function `f()` modifies its parameter `x` locally; the change does not affect the global variable `x`.
- The output will show the modified value of `x` inside the function, demonstrating local scope.
- The return value is captured by the variable `z`.
- After the function call, `x` remains 3 in the global scope, while `z` takes the value 4.

Fruitful Functions and Void Functions

- **Fruitful functions** specify a value to return using “return.”
- **Void functions** are called for their effects, such as printing or modifying data, but with no explicit return statement, they return None.

Fruitful Functions and Void Functions (example)

```
>>> def print_hello():  
...     print("Hello") # Void function  
...  
>>> def sum_two_numbers(a, b):  
...     return a + b # Fruitful function  
...  
>>> a = print_hello()  
Hello  
>>> print(a)  
None  
>>> b = sum_two_numbers(5, 3)  
>>> print(b)  
8
```