**Python Introduction**

**Install Python:**

* <https://www.programiz.com/python-programming/first-program>

**Keywords and Identifiers:**

* <https://www.programiz.com/python-programming/keywords-identifier>

**Comments:**

* <https://www.programiz.com/python-programming/comments>

**Variables, Constants and Literals:**

* <https://www.programiz.com/python-programming/variables-constants-literals>

**Data Types:**

* Since everything is an object in Python programming, data types are actually classes and variables are instances(object) of these classes.
* <https://www.programiz.com/python-programming/variables-datatypes>

**Type Conversion:**

* <https://www.programiz.com/python-programming/type-conversion-and-casting>

**Basic Input and Output:**

* <https://www.programiz.com/python-programming/input-output-import>

**Operators:**

* <https://www.programiz.com/python-programming/operators>

**Namespace and Scope:**

* <https://realpython.com/python-namespaces-scope/>
* <https://www.programiz.com/python-programming/namespace>
* In a Python program, there are four types of namespaces:
* Built-In
* Global
* Enclosing
* Local
* These have differing lifetimes.
* As Python executes a program, it creates namespaces as necessary and deletes them when they’re no longer needed. Typically, many namespaces will exist at any given time.
* **The Built-In Namespace:**
* The built-in namespace contains the names of all of Python’s built-in objects. These are available at all times when Python is **running**.
* Use **dir(\_\_builtins\_\_)** to list all the built-in namespace objects
* This namespace remains in existence until the interpreter terminates.
* **The Global Namespace:**
* The global namespace contains any names defined at the level of the main program. Python creates the global namespace when the main program body starts, and it remains in existence until the interpreter terminates.
* Strictly speaking, this may not be the only global namespace that exists. The interpreter also creates a global namespace for any module that your program loads with the import statement.
* <https://realpython.com/python-main-function>
* <https://realpython.com/python-modules-packages>
* **The Local and Enclosing Namespaces:**
* The interpreter creates a new namespace whenever a function executes.
* That namespace is local to the function and remains in existence until the function terminates.
* Enclosing namespace created when a function inside a function called.
* Each of these namespaces remains in existence until its respective function terminates. Python might not immediately reclaim the memory allocated for those namespaces when their functions terminate, but all references to the objects they contain cease to be valid.

**Variable Scope:**

* The existence of multiple, distinct namespaces means several different instances of a particular name can exist simultaneously while a Python program runs. As long as each instance is in a different namespace, they’re all maintained separately and won’t interfere with one another.
* But that raises a question: Suppose you refer to the name x in your code, and x exists in several namespaces. How does Python know which one you mean?
* The scope of a name is the region of a program in which that name has meaning.
* The interpreter determines this at runtime based on where the name definition occurs and where in the code the name is referenced.
* To return to the above question, if your code refers to the name x, then Python searches for x in the following namespaces in the order shown:
* **Local**: If you refer to x inside a function, then the interpreter first searches for it in the innermost scope that’s local to that function.
* **Enclosing**: If x isn’t in the local scope but appears in a function that resides inside another function, then the interpreter searches in the enclosing function’s scope.
* **Global**: If neither of the above searches is fruitful, then the interpreter looks in the global scope next.
* **Built-in**: If it can’t find x anywhere else, then the interpreter tries the built-in scope.
* If the interpreter doesn’t find the name in any of these locations, then Python raises a **NameError exception**.

**Python Namespace Dictionaries:**

* For global and local namespaces, that’s precisely what they are! Python really does implement these namespaces as dictionaries.
* Note: The built-in namespace doesn’t behave like a dictionary. Python implements it as a module.
* Python provides built-in functions called **globals()** and **locals()** that allow you to access global and local namespace dictionaries.
* The **globals()** function
* The built-in function globals() returns a reference to the current global namespace dictionary. You can use it to access the objects in the global namespace.
* You can create and modify entries in the global namespace using the globals() function as well:
* The **locals()** function
* Python also provides a corresponding built-in function called locals(). It’s similar to globals() but accesses objects in the local namespace instead:
* If you call locals() outside a function in the main program, then it behaves the same as globals().
* Deep Dive: A Subtle Difference Between globals() and locals()
* globals() returns an actual reference to the dictionary that contains the global namespace. That means if you call globals(), save the return value, and subsequently define additional variables, then those new variables will show up in the dictionary that the saved return value points to:
* locals(), on the other hand, returns a dictionary that is a current copy of the local namespace, not a reference to it. Further additions to the local namespace won’t affect a previous return value from locals() until you call it again. Also, you can’t modify objects in the actual local namespace using the return value from locals():
* Modify Variables Out of Scope:
* Sometimes a function can modify its argument in the calling environment by making changes to the corresponding parameter, and sometimes it can’t:
* An immutable argument can never be modified by a function.
* A mutable argument can’t be redefined wholesale, but it can be modified in place.
* A similar situation exists when a function tries to modify a variable outside its local scope. A function can’t modify an immutable object outside its local scope at all:
* The **global** and **nonlocal** keywords