CS303E: Elements of Computers and Programming

Simple Python

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: April 12, 2021 at 09:12

Assignment Statements

An assignment in Python has form:

Variable Name = Value

This means that variable is *assigned* **value**. I.e., after the assignment, **variable** "contains" **value**.

```
>>> x = 17.2
>>> y = -39
>>> z = x * y - 2
>>> print( z )
-672.8
```


Variables

A **variable** is a named memory location used to store values. We'll explain shortly how to name variables.

Unlike many programming languages, Python variables do not have associated types.

```
// C code
int x = 17;     // variable x has type int
x = 5.3;        // illegal
```

```
# Python code
x = 17          # x gets int value 17
x = 5.3         # x gets float value 5.3
```

A variable in Python actually holds a *pointer* to a class object, rather than the object itself.

Types in Python

Is it correct to say that there are no types in Python?

Yes and no. It is best to say that Python is "dynamically typed." Variables in Python are untyped, but values have associated types (actually classes). In some cases, you can convert one type to another.

Most programming languages assign types to both variables and values. This has its advantages and disadvantages.

Can you guess what the advantages are?

Variables and Assignments

You can create a new variable in Python by assigning it a value. *You don't have to declare variables, as in many other programming languages.*

```
>>> x = 3           # creates x, assigns int
>>> print(x)
3
>>> x = "abc"       # re-assigns x a string
>>> print(x)
abc
>>> x = 3.14        # re-assigns x a float
>>> print(x)
3.14
>>> y = 6           # creates y, assigns int
>>> x * y           # uses x and y
18.84
```


Meaning of a Variable

```
x = 17              # Defines and initializes x
y = x + 3           # Defines y and initializes y
z = w               # Runtime error if w undefined
```

This code defines three variables x, y and z. Notice that on the *left hand side* of an assignment the variable is created (if it doesn't already exist), and given a value. On the lhs, it stands for a *location*.

On the *right hand side* of an assignment, it stands for the current *value* of the variable. If there is none, it's an error.

Naming Variables

Below are (most of) the rules for naming variables:

- Variable names must begin with a letter or underscore ("_") character.
- After that, use any number of letters, underscores, or digits.
- Case matters: "score" is a different variable than "Score."
- You can't use *reserved words*; these have a special meaning to Python and cannot be variable names.

Naming Variables

Python Reserved Words:

and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, nonlocal, None, not, or, pass, raise, return, True, try, while, with, yield

IDLE and many IDEs display reserved words in color to help you recognize them.

Not Reserved, but Don't Use

Function names like `print` are *not* reserved words. But using them as variable names is *a very bad idea* because it redefines them.

```
>>> x = 17
>>> print(x)
17
>>> print = 23
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```


Naming Variables

```
>>> ___ = 10                        # wierd but legal
>>> _123 = 11                       # also wierd
>>> ab_cd = 12                      # perfectly OK
>>> ab|c = 13                       # illegal character
  File "<stdin>", line 1
SyntaxError: can't assign to operator
>>> assert = 14                     # assert is reserved
  File "<stdin>", line 1
    assert = 14
           ^
SyntaxError: invalid syntax
>>> maxValue = 100                  # good one
>>> print = 8                       # legal but ill-advised
>>> print( "abc" )                  # we've redefined print
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```


Naming Variables

In addition to the rules, there are also some conventions that good programmers follow:

- Variable names should begin with a lowercase letter.
- Choose meaningful names that describe how the variable is used. This helps with program readibility.

  Use `max` rather than `m`.
  Use `numberOfColumns` rather than `c`.
- One exception is that loop variables are often i, j, etc.

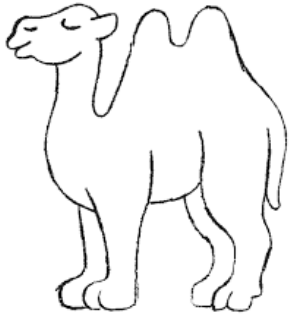
```
    for x in lst: print( x )
```

rather than:

```
    for listItem in lst: print( listItem )
```

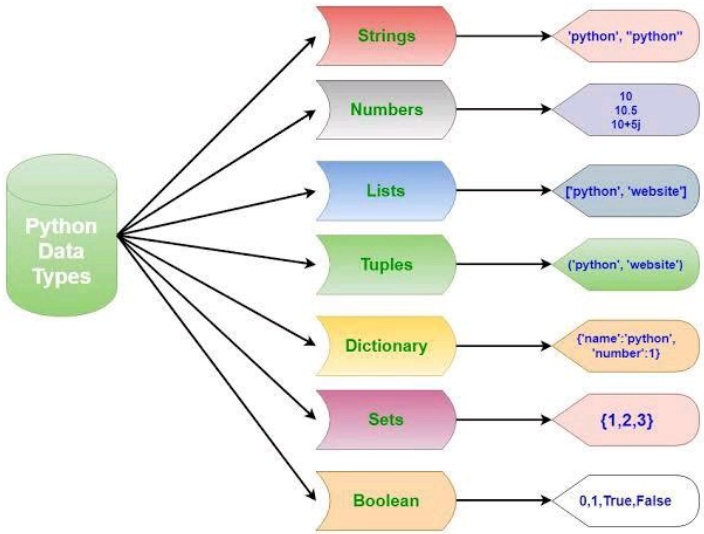

Camel Casing

If you use a multi-word names (good practice), use "camel casing": `avgHeight`, `countOfItems`, etc.



These are just conventions; you'll see lots of counterexamples in real code.

Common Python Data Types



What is a Data Type?

A **data type** is a kind of value.

Type	Description	Syntax example
int	An immutable fixed precision number of unlimited magnitude	42
float	An immutable floating point number (system-defined precision)	3.1415927
str	An immutable sequence of characters.	'Wikipedia' "Wikipedia" """Spanning multiple lines"""
bool	An immutable truth value	True, False
tuple	Immutable, can contain mixed types	(4.0, 'string', True)
bytes	An immutable sequence of bytes	b'Some ASCII' b"Some ASCII"
list	Mutable, can contain mixed types	[4.0, 'string', True, 4.0]
set	Mutable, unordered, no duplicates	{4.0, 'string', True}
dict	A mutable group of key and value pairs	{'key1': 1.0, 3: False}

The type Function

```
>>> x = 17
>>> type(x)
<class 'int'>
>>> y = -20.9
>>> type(y)
<class 'float'>
>>> type(w)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'w' is not defined
>>> lst = [1, 2, 3]
>>> type(lst)
<class 'list'>
>>> type(20)
<class 'int'>
>>> type( (2, 2.3 ) )
<class 'tuple'>
>>> type('abc')
<class 'str'>
>>> type( {1, 2, 3} )
<class 'set'>
>>> type(print)
<class 'builtin_function_or_method'>
```

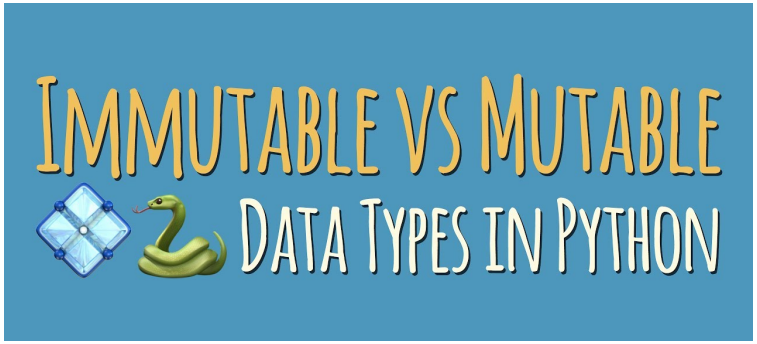

Three Common Data Types

Three data types you'll encounter in many Python programs are:

- `int`: signed integers (whole numbers)
  - Computations are exact and *of unlimited size*
  - Examples: 4, -17, 0
- `float`: signed real numbers (numbers with decimal points)
  - Large range, but fixed precision
  - Computations are approximate, not exact
  - Examples: 3.2, -9.0, 3.5e7
- `str`: represents text (a string)
  - We use it for input and output
  - We'll see more uses later
  - Examples: "Hello, World!", 'abc'

These are all *immutable*.

Mutable vs. Immutable



An **immutable** object is one that cannot be changed by the programmer after you create it; e.g., numbers, strings, etc.

A **mutable** object is one that can be changed; e.g., sets, lists, etc.

What Immutable Means

An **immutable** object is one that cannot be changed by the programmer after you create it; e.g., numbers, strings, etc.

It also means that *there is only one copy of the object in memory.* Whenever the system encounters a new reference to 17, say, it creates a pointer to the already stored value 17.

Every reference to 17 is actually a pointer to the *only* copy of 17 in memory. Ditto for `"abc"`.

If you do something to the object that yields a new value (e.g., uppercase a string), you're actually creating a new object, not changing the existing one.

Immutability

```
>>> x = 17           # x holds a pointer to the object 17
>>> y = 17           # so does y
>>> x is y           # x and y point to the same object
True
>>> id(x)            # the unique id associated with 17
10915008
>>> id(y)
10915008
>>> s1 = "abc"       # creates a new string
>>> s2 = "ab" + "c"  # creates a new string (?)
>>> s1 is s2         # actually it doesn't!
True
>>> id(s1)
140197430946704
>>> id(s2)
140197430946704
>>> s3 = s2.upper()     # uppercase s2
>>> print(s3)
ABC
>>> id(s3)           # this is a new string
140197408294088
```

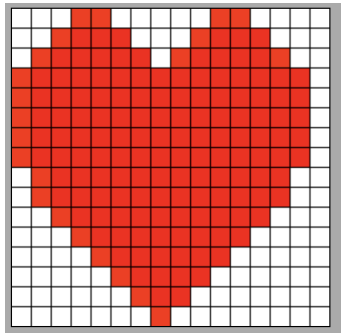

Let's Take a Break

How is Data Stored?

Fundamental fact: *all data* in the computer is stored as a series of bits (0s and 1s) in the memory.

That's true whether you're storing numbers, letters, documents, pictures, movies, sounds, programs, etc. *Everything!*

A key problem in designing any computing system or application is deciding how to *represent* the data you care about as a sequence of bits.



How is Data Stored: Digital Images

For example, images can be stored digitally in any of the following formats (among others):

- JPEG: Joint Photographic Experts Group
- PNG: Portable Network Graphics
- GIF: Graphics Interchange Format
- TIFF: Tagged Image File
- PDF: Portable Document Format
- EPS: Encapsulated Postscript

Most of the time, we won't need to know how data is stored in the memory. The computer will take care of that for you.

How is Data Stored?

The memory can be thought of as a big array of **bytes**, where a byte is a sequence of 8 bits. Each memory address has an **address** (0..maximum address) and **contents** (8 bits).

...	...	
...	...	
10000	01001010	Encoding for character 'J'
10001	01100001	Encoding for character 'a'
10002	01110110	Encoding for character 'v'
10003	01100001	Encoding for character 'a'
...	...	
...	...	

A byte is the smallest unit of storage a programmer can address. We say that the memory is *byte-addressable*.

Representation Example: ASCII

The standard way to represent *characters* in memory is ASCII. The following is part of the ASCII (American Standard Code for Information Interchange) representation:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	—	}		

The standard ASCII table defines 128 character codes (from 0 to 127), of which, the first 32 are control codes (non-printable), and the remaining 96 character codes are printing characters.

How is Data Stored

Characters or small numbers can be stored in one byte. If data can't be stored in a single byte (e.g., a large number), it must be split across a number of adjacent bytes in memory.

The way data is encoded in bytes varies depending on:

- the data type
- the specifics of the computer

Most of the time, we won't need to know how data is stored in the memory. The computer will take care of that for you.

Formats of Data Types

It would be nice to look at the character string "25" and do arithmetic with it.

However, the `int` 25 (a number) is represented in binary in the computer by: `00011001`. Why?

And the string "25" (two characters) is represented by: `00110010 00110101`. Why?

`float` numbers are represented in an even more complicated way, since you have to account for an exponent. (Think "scientific notation.") So the number 25.0 (or $2.5 * 10^1$) is represented in yet a third way.

Data Type Conversion

Python provides functions to *explicitly* convert numbers from one type to another:

`float` (< number, variable, string >)
`int` (<number, variable, string >)
`str` (<number, variable >)

Note: `int` *truncates*, meaning it throws away the decimal point and anything that comes after it. If you need to *round* to the nearest whole number, use:

`round` (<number or variable >)

Conversion Examples

```
float(17)
17.0
>>> str(17)
'17'
>>> int(17.75)                          # truncates
17
>>> str(17.75)
'17.75'
>>> int("17")
17
>>> float("17")
17.0
>>> round(17.1)
17
>>> round(17.6)
18
round(17.5)                           # round to even
18
>>> round(18.5)                       # round to even
18
```

Why does round to even make sense?

Conversion Examples

If you have a string that you want to (try to) interpret as a number, you can use `eval`.

```
>>> eval("17")
17
>>> eval("17 + 3")
20
>>> eval(17 + 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: eval() arg 1 must be a string,
   bytes or code object
```

What was wrong with the last example?

Be Cautious Using `eval`

Using the function `eval` is considered dangerous, especially when applied to user input.

`eval` passes its argument to the Python interpreter, and a malicious (or careless) user could input a command string that could:

- delete all of your files,
- take over your machine, or
- some other horrible thing.

Use `int()` or `float()` is you want to convert a string input into one of these types.

Arithmetic Operations

Here are some useful operations you can perform on numeric data types.

Name	Meaning	Example	Result
+	Addition	34 + 1	35
-	Subtraction	34.0 - 0.1	33.9
*	Multiplication	300 * 30	9000
/	Float division	1 / 2	0.5
//	Integer division	1 // 2	0
**	Exponentiation	4 ** 0.5	2.0
%	Remainder	20 % 3	2

`(x % y)` is often referred to as "x mod y".

Simple Program: Pythagorean Triples

In file `pythagoreanTriple.py`:

```
""" The sides of a right triangle satisfy the relation:
      a**2 + b**2 = c**2.
    Test whether the three integers in variables a, b, c
    form a pythagorean triple, i.e., satisfy this relation.
       """

a = 3
b = 4
c = 5
ans = ( a**2 + b**2 == c**2 )
print("a:", a, "b:", b, "c:", c, \
      "is" if ans else "is not", \
      "a pythagorean triple" )
```

```
> python pythagoreanTriple.py
a: 3 b: 4 c: 5 is a pythagorean triple
```

Augmented Assignment Statements

Python (like C) provides a shorthand syntax for some common assignments:

```
i += j     means the same as     i = i + j
i -= j     means the same as     i = i - j
i *= j     means the same as     i = i * j
i /= j     means the same as     i = i / j
i //= j    means the same as     i = i // j
i %= j     means the same as     i = i % j
i **= j    means the same as     i = i ** j
```

```
>>> x = 2.4
>>> x *= 3.7          # same as x = x * 3.7
>>> print(x)
8.88
```


Mixed-Type Expressions

Most arithmetic operations behave as you would expect for numeric data types.

- Combining two floats results in a float.
- Combining two ints results in an int (except for /). Use // for integer division.
- Dividing two ints gives a float. E.g., `2 / 5` yields 2.5.
- Combining a float with an int usually yields a float.

Python will figure out what the result should be and return a value of the appropriate data type.

Mixed Type Expressions

```
>>> 5 * 3 - 4 * 6      # (5 * 3) - (4 * 6)
-9
>>> 4.2 * 3 - 1.2
11.400000000000002     # approximate result
>>> 5 // 2 + 4         # integer division
6
>>> 5 / 2 + 4          # float division
6.5
```


Special Assignment Statements

Simultaneous assignments:

```
m, n = 2, 3
```

means the same as:

```
m = 2
n = 3
```

With the caveat that these happen *at the same time*.

What does the following do?

```
i, j = j, i
```

Multiple assignments:

```
i = j = k = 1
```

means the same as:

```
k = 1
j = k
i = j
```

Note that these happen right to left.

Advice on Programming

Think before you code!
Think before you code!
Think before you code!

- Don't jump right into writing code.
- Think about the overall process of solving your problem; write it down.
- Refine each part into subtasks. Subtasks may require further refinement.
- Code and test each subtask before you proceed.
- Add `print` statements to view intermediate results.

Advice on Programming

The software development process outlined in Section 2.13 is called *the waterfall model*. You'll do well to follow it, except on the simplest programs.