DSCI 510
PRINCIPLES OF PROGRAMMING FOR DATA SCIENCE

Itay Hen

PYTHON: TYPES (CONTINUED)

What Does "Type" Mean?

- In Python variables, literals, and constants have a "type"
- Python knows the difference between an integer number and a string
- For example "+" means "addition" if something is a number and "concatenate" if something is a string

```
>>> ddd = 1 + 4
>>> print(ddd)
5
>>> eee = 'hello ' + 'there'
>>> print(eee)
hello there
```

concatenate = put together

Type defines structure of values and allowed operations on them
Polymorphism: when the same operator behaves differently when applied to different types

Several Types of Numbers

- Numbers have two main types
  - Integers are whole numbers: -14, -2, 0, 1, 100, 401233
  - Floating Point Numbers have decimal parts: -2.5 , 0.0, 98.6, 14.0
- There are other number types - they are variations on float and integer

```
>>> xx = 1
>>> type (xx)
<class 'int'>
>>> temp = 98.6
>>> type(temp)
<class'float'>
>>> type(1)
<class 'int'>
>>> type(1.0)
<class'float'>
>>>
```

https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex

Type Conversions

- When you put an integer and floating point in an expression, the integer is implicitly converted to a float
- You can control this with the built-in functions int() and float()

```
>>> print(float(99) + 100)
199.0
>>> i = 42
>>> type(i)
<class'int'>
>>> f = float(i)
>>> print(f)
42.0
>>> type(f)
<class'float'>
>>>
```

String Conversions

- You can also use int() and float() to convert between strings and integers
- You will get an error if the string does not contain numeric characters

```
>>> sval = '123'
>>> type(sval)
<class 'str'>
>>> print(sval + 1)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object
to str implicitly
>>> ival = int(sval)
>>> type(ival)
<class 'int'>
>>> print(ival + 1)
124
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
with base 10: 'x'
```

Types and Conversion

	Type	Examples	Conversion Function	Value after conversion
Integer Numbers	int	32 -100	int(2.7182818) int('-212')	2 -212
Floating Point Numbers	float	3.14159 212.00	float(3) float('212.00')	3.0 212.0
Strings	str	'apple' '212.00'	str(3) str(212.00)	'3' '212.0'

Function **`type()`** returns the type of the argument:
e.g. `type(3.0)` → `<class 'float'>`

Python 3 Type Size Limits

- Integer: Arbitrary size!
  - Limited only by the available memory
  - Arbitrary-precision arithmetic
  - In Python 2 and other languages the built-in integers generally have a fixed size, e.g., 32 or 64 bits.
- Float: 64-bit floating point number (C double)
  - Largest float value ~ $1.79\ 10^{308}$
    - Larger numbers are infinite: sys.float_info.max * 2 == float('inf')
  - Smallest float value ~ $2.22\ 10^{-308}$

inf: unbounded upper value for comparison

```
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53,
epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

- Strings, Dictionaries, …
  - Limited by the largest index value
  - sys.maxsize = $2^{63} - 1$ = 9223372036854775807

Python is dynamically-typed

Python is a *dynamically-typed* language:

- No need to declare the type of the values of a variable
- The type of variables is checked at runtime
- Can reassign values of different types to the same variable

```
>>> x = 1
>>> print(x)
1
>>> x = 'Hi!'
>>> print(x)
Hi!
```

Statically-typed languages declare types of variables in the code and check them at compile-time:

- This will throw an error in Java:

```
String name = "John";
name = 15
```

Common Python Types

We'll discuss most of these at length later on

Basic Types

- `int` – integers (e.g., `1`, `-42`, `0`)
- `float` – floating-point numbers (e.g., `3.14`, `-2.7`)
- `complex` – complex numbers (e.g., `2+3j`)

Text Type

- `str` – strings of text (e.g., `"hello"`, `'Python'`)

Sequence Types

- `list` – ordered, mutable collection (e.g., `[1, 2, 3]`)
- `tuple` – ordered, immutable collection (e.g., `(1, 2, 3)`)
- `range` – arithmetic progression of integers (e.g., `range(5)` → `0,1,2,3,4`)

Mapping Type

- `dict` – key–value pairs (e.g., `{"a": 1, "b": 2}`)

Set Types

- `set` – unordered collection of unique elements (e.g., `{1, 2, 3}`)
- `frozenset` – immutable set

Boolean Type

- `bool` – truth values (`True`, `False`)

Special Type

- `NoneType` – represents "nothing" or absence of value (`None`)

User Input

- We can instruct Python to pause and read data from the user using the input() function
- The input() function returns a string

```
nam = input('Who are you? ')
print('Welcome', nam)
```

```
Who are you? Chuck
Welcome Chuck
```

What could possibly go wrong? Python Types

Consider this Python program:

```
n_adults = input('Enter number of adults: ')
n_children = input('Enter number of children: ')
print('Total people:', n_adults + n_children)
```

What will it print for inputs 2 (adults) and 3 (children)?

```
Total people: 23
```

Why? input() returns a **string** and '2' + '3' == '23' ('+' concatenates strings)

What the programmer probably meant:

```
n_adults = int(input('Enter number of adults: '))
n_children = int(input('Enter number of children: '))
print('Total people:', n_adults + n_children)
Total people: 5
```

Converting User Input

- If we want to read a number from the user, we must convert it from a string to a number using a type conversion function
- Later we will deal with bad input data

```
inp = input('Europe floor?')
usf = int(inp) + 1
print('US floor', usf)
```

Europe floor? 0
US floor 1

Comments in Python

- Anything after a # is ignored by Python
- Why comment?
  - Describe what is going to happen in a sequence of code
  - Document who wrote the code or other ancillary information
  - Turn off a line of code - perhaps temporarily

Comments

- Use comments
- No, really. Use comments!
- Make them meaningful
  - `vel = 5 # set vel to 5` **unhelpful comment**
  - `vel = 5 # Velocity in meters/second`

Example of a Nicely Commented Program

```
# Get the name of the file and open it
name = input('Enter file:')
handle = open(name, 'r')

# Count word frequency
counts = dict()
for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word,0) + 1

# Find the most common word
bigcount = None
bigword = None
for word,count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

# All done
print(bigword, bigcount)
```

Comments (2)

- Docstrings, denoted by the triple quotes, e.g.:

```
"""function(a, b) -> list"""
```

- https://peps.python.org/pep-0257/
- Use multiple lines to write a longer comment
- High-level explanation of what the function does
- List all required input parameters and their types
- Describe what the output would look like

```
def add(a, b):
    """
    Add two numbers together.

    Parameters:
        a (int or float): The first number.
        b (int or float): The second number.

    Returns:
        int or float: The sum of a and b.
    """
    return a + b
```

OPERATORS IN PYTHON

Operators in Python

- Arithmetic Operators (+, -, /, *, %, **, //)
- Assignment Operators (=, +=, -=, *=, /=, %=, **=, //=)
- Comparison Operators (==, !=, <, >, <=, >=)
- Identity Operators (is, is not)
- Logical Operators (AND, OR, NOT)
- Bitwise Operators (&, |, ^, ~, <<, >>)
- Membership Operators (in, not in)

Arithmetic Operators

Operator	Example
Addition	15 + 4 = 19
Subtraction	15 - 4 = 11
Multiplication	15 * 4 = 60
Division	15 / 4 = 3.75
Exponentiation	15 ** 4 = 50625
Modulus	15 % 4 = 3
Floor Division	15 // 4 = 3

Assignment Operators

Operator	Example	Equivalent to	Value of x
=	x = 56	x = 56	56
+=	x += 3	x = x + 3	59
-=	x -= 3	x = x - 3	56
/=	x /= 4	x = x / 4	14.0
*=	x *= 5	x = x * 5	70.0
**=	x **= 2	x = x ** 2	4900.0
%=	x %= 458	x = x % 458	320.0
//=	x //= 34	x = x // 34	9.0

Variables / Assignment

- Variables are containers for storing values
- Assignment:

  Variable = Expression

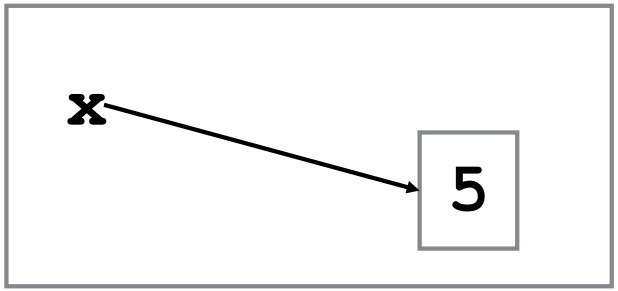
  pi = 3.141592653589793

  H2O = "water"

  gravity_on_earth = 9.80665 # in m/s^2

What Does Assignment "=" Really Mean ?


```
>>> x = 5
```



- What happens when we execute this?
- Python:
  - Creates a variable called '**x**', if it doesn't already exist,
  - Makes it 'point' to specific location ("address") in memory, and
  - places the integer **5** at that address.
  - If we assign a value of a different type (e.g. a string), Python automatically changes the size of the memory needed to hold that value

Parallel Assignments

- Multiple variables are assigned expressions at once
- **All** expressions are computed **before** any are assigned

```
x, y, z = 2, 5.0, "Python"
```
x is 2, y is 5.0, z is “Python”

```
x, y = y, x
```
x and y are swapped: x is 5.0, y is 2

```
y, z = x**y, y *z
```
y is 25.0, z is “PythonPython”

For comparison, the following does not swap x and y

```
x, y = 2, 5.0
```
x is 2, y is 5.0

```
x = y
```
x is 5.0, y is 5.0

```
y = x
```
x is 5.0, y is 5.0

Diversity of + Operator

```
y, z = [1, 2, 3], "cat"
```

y is a list: [1, 2, 3], z is “cat"

```
y += [55, 66]
```

y is a list: [1, 2, 3, 55, 66]

```
z += "nap"
```

z is “catnap”

*We will cover lists at length later

Comparison and Identity Operators

Operator	Meaning
x == y	is equal to
x != y	is not equal to
x > y	greater than
x < y	lesser than
x >= y	greater than or equal to
x <= y	lesser than or equal to
x is y	is the same as
x is not y	is not the same as

- Comparison operators operate on operands which are Python constants, variables of expressions
- Each of the above is a “Boolean Expression”
- Boolean expressions ask a question, to which the answer is “yes” or “no”, ie, **True** or **False**

Boolean Variables

Boolean (bool) is another type of variable (like int, float, str), but it can only take two values:

- True
- False (note capitalization!)

```
>>> type(True)
<type 'bool'>
```

By the way, what are the outputs of the following?


```
>>> type("True")
>>> type(true)
```

Equality is Not Assignment

- These are not the same:

```
my_var = 10
my_var == 10
```

- If you confuse the two, Python will either let you know (e. g., it's syntax error), or your program will fail in another way…

Choosing between == and is

- == compares the values of objects (the data they hold), while is compares their identities.
- While programming we often care about values than object identities, so == appears more frequently than is in Python code
- **is** checks identity: whether two variables point to the same object in memory.
- **==** checks equality → whether the values are the same.

Equality of values: '==' Equality of memory locations: 'is'

```
>>> x = [1, 2]
>>> y = [1, 2]
>>> x is y

>>>False

>>> x == y

True

>>> y = x
>>> x is y

True

>>> x[0] = 'a'
>>> y

['a', 2]
```

However, for small integers [-5 … 256]:

```
>>> x = 2
>>> y = 2
>>> x == y
True
>>> x is y
True
>>> x = 257
>>> y = 257
>>> x == y
True
>>> x is y
False
```

Don't use `is` if you just need `==`
You are asking for trouble!

Logical Operators

- *Logical operators* operate on Boolean expressions/values
  - On things like (x == 'Bobby'), False, (y > 2)
- There are three logical operators:
  - and
  - or
  - not
- Their meaning in Python is essentially their common-sense meaning
- Are you familiar with "truth tables"?

Truth Tables

‘and’	True	False
True	True	False
False	False	False

‘or’	True	False
True	True	True
False	True	False

‘not’	
True	False
False	True

‘and’	1	0
1	1	0
0	0	0

‘or’	1	0
1	1	1
0	1	0

‘not’	
1	0
0	1

Logical Operator Caveats

- You can plug just about anything into a logical operator
  - For example, nonzero integers will evaluate to True

```
>>> x = 7
>>> x and (1 / x < 1)
>>> True
```

- <u>But you do not want to do this!</u>
  - Stick with things you know evaluate explicitly to Boolean variables (ie Boolean expressions)

```
>>> (x != 0) and ( 1 / x < 1)
>>> True
```

Evaluation inside Boolean Expressions

Type	False	True
int	0	other, eg: 123
float	0.0	other, eg: 0.1
str	“”	other, eg ‘a’
list	[]	other, eg: [0]
dict	{}	other, eg: {‘count’: 0}
bool	False	True

Bitwise Operators

Operator	Example	Explanation
Bitwise AND (&) Performs bitwise AND operation on corresponding bits of two integers Resulting bits is 1 only if both bits are 1	5 & 3 = 1	5 in binary: 0101 3 in binary: 0011 Result: 0001 (1 in decimal)
Bitwise OR (|) Performs bitwise OR operation on corresponding bits of two integers. Resulting bit is 1 if at least one of the bits is 1	10 | 7 = 15	10 in binary: 1010 7 in binary: 0111 Result: 1111 (15 in decimal)
Bitwise XOR (^) Performs bitwise exclusive OR operation on corresponding bits of two integers Resulting bit is 1 if exactly one of the bits is 1	8 ^ 12 = 4	8 in binary: 1000 12 in binary: 1100 Result: 0100 (4 in decimal)
Bitwise NOT (~) Inverts all the bits of an integer, changing 0s to 1s and vice versa	~15 = -16	15 in binary: 1111 Result: -16 (Due to two's complement representation)
Left Shift (<<) Shifts the bits of an integer to the left by a specified number of positions. New bits on the right are filled with zeros	1 << 2 = 4	1 in binary: 0001 Result: 0100 (4 in decimal)
Right shift (>>) Shifts the bits of an integer to the right by a specified number of positions For positive numbers, new bits on the left are filled with zeros	16 >> 3 = 2	16 in binary: 10000 Result: 00010 (2 in decimal)

Two's complement

Two's complement is the most common way computers represent **signed integers** (positive and negative whole numbers) in binary.

Key ideas:

1. **Positive numbers** look the same as in normal binary.
   - Example: $+5$ in 8 bits → `00000101`.
2. **Negative numbers** are stored by taking the binary of the absolute value, flipping all the bits (one's complement), and then **adding 1**.
   - Example: To represent -5 in 8 bits:
     - Start with $+5$: `00000101`
     - Flip the bits: `11111010` (one's complement)
     - Add 1: `11111011` → this is -5.
3. **Range**: For an n-bit two's complement system, integers run from -2^{n-1} to $2^{n-1} - 1$.
   - Example: With 8 bits → range is -128 to $+127$.
4. **Why use it?**
   - Addition and subtraction work the same way for positive and negative numbers (no special rules needed).
   - There is only one zero (`0000...0000`), unlike one's complement which has both a positive and negative zero.

Membership Operators

- Membership operators are used to test whether a value is present in a sequence, such as a list, tuple or string
- They return a Boolean value (True or False) based on the presence or absence of the value in the sequence.

```
>>> ‘apple’ in [‘apple’, ‘banana’, ‘orange’]
>>> True
>>> ‘grape’ not in [‘apple’, ‘banana’, ‘orange’]
>>> True
>>> ‘world’ in ‘Hello world!’:
>>> True
```

```
>>> ‘age’ in {‘name’: ‘Alice’, ‘age’: 30}
>>> True
```

PYTHON: STRINGS AND PRINTING

Special Characters

Escape Sequence	What it Does
\\	Backslash (\)
\’	Single-quote(‘)
\”	Double-quote(“)
\a	ASCII bell (BEL)
\b	ASCII backspace
\n	ASCII linefeed (LF)
\r	Carriage Return (CR)
\t	Horizontal Tab (TAB)

Printing: Escape Character

- When printing we surround the text with quote marks (')
- We can also use double quotes (")
- What happens if we need to print this line?

  I am 5'9" tall

  print(" I am 5'9\" tall")
- And what if we want to print this line?

  I want to see the 5'9\" backslash

  print(" I want to see the 5'\\\" backslash ")

Printing: Formatted Strings

Formatted Printing with {}

```
>>> hours = 20
>>> rate = 17.5
>>> print('If I work', hours, 'hours at' , rate, "per hour, I'll make $", hours*rate)
If I work 20 hours at 17.5 per hour, I'll make $350.0

>>> print(f"If I work {hours} hours at {rate} per hour, I'll make ${hours*rate}")
If I work 20 hours at 17.5 per hour, I'll make $350.0

>>> print(f'If I work {hours} hours at {rate} per hour, I\'ll make ${hours*rate}')
If I work 20 hours at 17.5 per hour, I'll make $350.0

>>> print(f"If I work {hours} hours at {rate} per hour, \
... I'll make ${hours*rate}")
If I work 20 hours at 17.5 per hour, I'll make $350.0
```

Printing: Triple Quotes

- Printing strings with new lines:
  enclose in triple single or double quotes

```
print("""
A short story
about the
ultimate
question
about the
universe
...
""")
```