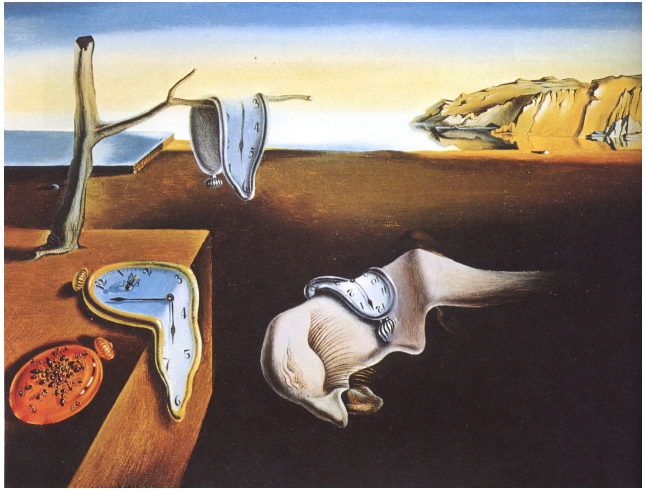


JamCoders: Week 1

Lecture 1B:

- Data Types
- (more) operators
- casting

“Abstraction”

Abstraction: thinking with **ideas** instead of **concrete details**

- For us, abstraction means that we can think on a higher level than binary

- Instead of having to write 0000 0101, every time we want to have the number five, we can just write 5

```
my_number = 5
print(my_number)
```

Data Types

In order to abstract away how the computer is storing things, we use different data types.

The compiler will treat different data types differently

```
a = 9 + 7
print(a)
b = "9" + "7"
print(b)
```

Data Types

There are 13 built-in datatypes in Python, we mainly use these 9:

- booleans, integers, floats, strings
- lists, tuples, ranges
- dicts, sets

Data Types

There are 13 built-in datatypes in Python, we mainly use these 9:

- **booleans, integers, floats, strings**
- lists, tuples, ranges
- dicts, sets

Today, we will learn about the first 4. You'll learn about the others over the next few weeks.

Booleans

Integers

Floats

Strings

Booleans

- Booleans can be one of two options: True or False
- Can be thought of as “is” types of questions: “Is it true that the state of my program is x?”

Is it true that the bison is wearing a hat?

Booleans

Is it true that the bison is wearing a hat?

```
is_wearing_hat = False
```

```
is_wearing_hat = True
```

Booleans

Boolean True and False must be capitalized!

```
is_wearing_hat = False
```

```
is_wearing_hat = True
```

Booleans

Boolean True and False must be capitalized!

```
is_wearing_hat = False
```

```
is_wearing_hat = True
```

How do we use booleans to do useful things?

Operators

Operators

Operators are special symbols that are used to perform operations on values and variables.

A few different kinds that we use all of the time:

- boolean operators
- arithmetic operators
- comparison operators
- string operators

As we'll see, the same *operator* will do a different thing depending on the data type.

Boolean Operators

- **not** flips the boolean from one value to the opposite

```
is_absent = not is_in_class
```

Boolean Operators

- **not** flips the boolean from one value to the opposite

```
is_absent = not is_in_class
```

- **and** evaluates to **True** if *both* sides of the expression are **True**

```
can_buy_beer = has_funds and has_adult_id
```

Boolean Operators

- **not** flips the boolean from one value to the opposite

```
is_absent = not is_in_class
```

- **and** evaluates to **True** if *both* sides of the expression are **True**

```
can_buy_beer = has_funds and has_adult_id
```

- **or** evaluates to **True** if *either* side of the expression is **True**

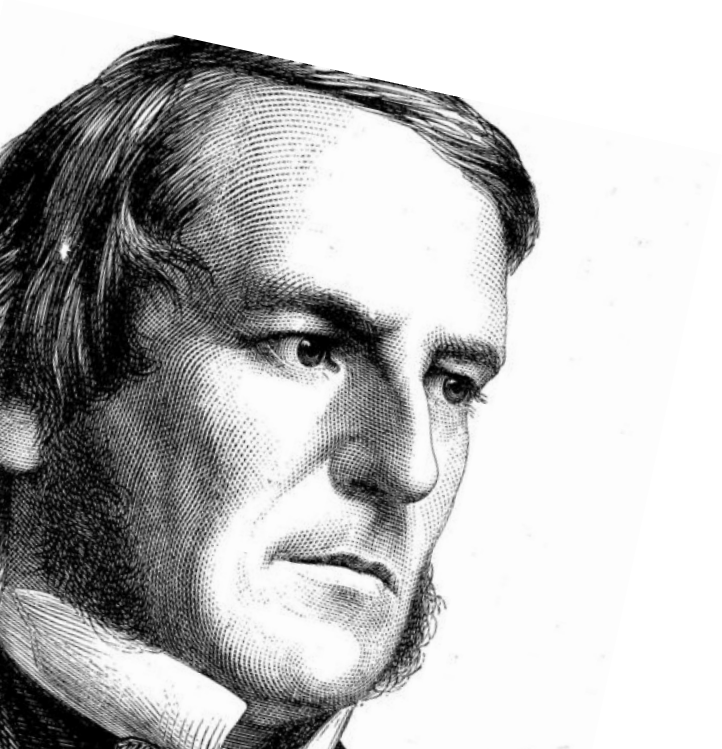
```
has_adult_id = is_over_18 or has_fake_id
```

Common Questions

Q: Why are these called booleans?

Common Questions

Q: Why are these called booleans?



George Boole was a British scientist who discovered algebra of logic - or using variables to represent True or False.

Booleans

Integers

Floats

Strings

Integers

Integers are whole numbers.

How many bunnies?

Integers can be positive, negative, or 0.

```
num_bunnies = 2
```

Review: Arithmetic Operators

The same operators that you're used to in math work in Python.

- Addition (+)

```
sum = 5 + 6
```

- Subtraction (-)

```
diff = 12 - 5
```

- Multiplication (*)

```
product = 8 * 4
```

- Division (/)

```
quot = 12 / 5
```

- Exponentiation (**)

```
power = 2 ** 6
```

(this is the same as 2^6)

Review: Arithmetic Operators

If applying multiple operators, use parentheses to control the order they get evaluated, just like in math:

```
my_variable = 7 + ((20 / 4) ** 2)
print(my_variable)
```

my_variable = 7 + ((20 / 4) ** 2)

my_variable = 7 + (5.0 ** 2)

my_variable = 7 + 25.0

my_variable = 32.0

Comparison Operators

It can be handy to be able to compare values. Python to the rescue…

All of the comparison operators evaluate to a boolean, **True** or **False**

Operator	Name	Example
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b
==	Equal to	a == b
!=	Not equal to	a != b

Comparison Operators

It can be handy to be able to compare values. Python to the rescue…

All of the comparison operators evaluate to a boolean, **True** or **False**

Operator	Name	Example
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b
==	Equal to	a == b
!=	Not equal to	a != b

Common mistake is to accidentally use one equals sign, =, which is used for variable assignment, instead of two equals signs, ==, for comparison.

Booleans

Integers

Floats

Strings

Floats

But what about representing numbers that aren't whole numbers?

- Money ($4.99)
- Percentages and fractions (12%, 0.12)
- GPA (3.51)

We use **floats** to represent **real** numbers.

Arithmetic Operators

Float operators work the same way as int operators.

You can **mix and match** ints and floats, the result will always be a float.

```
summation = 5 + 6.1
print(summation)
```

```
user@mimir: ~/csci_1
 python3 scratch.py
11.1
```

Floats

Commonly Asked Questions

Q: *Why* is it called a float? Why not just call it a real number?

A: Take a Computer Architecture course and you'll find out!

Q: When I do float arithmetic and print the result, I get a weird number? Did I do something wrong?

A: Representing fractions and decimals in binary is **hard**. Sometimes, the results will be off by a tiny bit (imprecise). That's normal.

Booleans

Integers

Floats

Strings

Strings

- A *character* is a single letter or symbol, like “c” or “!”
- A *string* is a sequence of characters

In Python, anything within quotation marks is a string and is taken literally:

```
x = "hello"
```

You can use double (“ ”) or single (’ ’) quotation marks, just be consistent.

String Operators

Concatenating Strings: + (add)

```
name = "Frederick”
greeting = "Hello, “ + name
```

Repeating a string: * (multiply)

```
laugh = "ha”
laugh_harder = laugh * 3
print(laugh_harder)
```

Prints “hahaha”

String Comparison Operator

To determine if two strings are equal, use the **==** operator. Note that it is case sensitive.

```
first_school_name = "Berkeley"
second_school_name = "Stanford"
print(first_school_name == second_school_name)
```

Prints False, because the strings are different.

String Length

This one isn't an operator, but is still useful to know:

- **len()** - given a string, returns the length of that string

```
length = len("Bison")
```

Working with Types

Casting

Casting is how we converting between different types.

You **cast** a variable or value of one type to another type by using the function of the destination type:

- **int()** to convert something to an integer
- **str()** to convert something to a string
- **float()** to convert something to a float
- **bool()** to convert something to a boolean

Casting

String to int/float or vice versa is most common

- Use int(“8”) to convert the string “8” to the integer 8.
- Use str(8) to convert the integer 8 to the string “8”

```
num_of_apples = "9"
num_of_bananas = "7"

print(int(num_of_apples) + int(num_of_bananas))
```

Finding the type of a variable

We can use the **type()** function to find out the type of any variable.

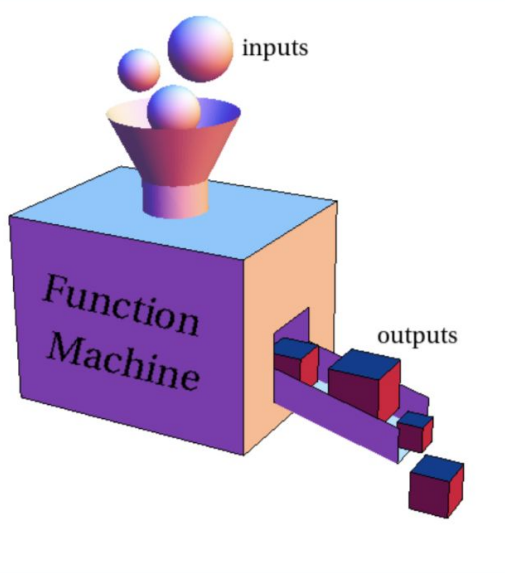
```
var1 = "Alex"
var2 = True
var3 = 12
var4 = 7.5
print(type(var1))
print(type(var2))
print(type(var3))
print(type(var4))
```

```
user@mimir: ~/csci_100_h
 python3 types.py
<class 'str'>
<class 'bool'>
<class 'int'>
<class 'float'>
```

Expressions

Functions

Functions take input, perform some task, and optionally give an output.



Function calls are also expressions, and can be used as operands.

Examples

- **str()** is a function that evaluates to a string
- **int()** is a function that evaluates to an integer.

We'll learn much more about functions on day 3. For now, just know that function calls are expressions.

Expressions

We stop evaluating expressions once they simplify down to a single value, known as an “atom”, which can either be an *identifier* (variable name) or a *literal*.

```
"Result: " + str((2 ** 6) / 4)
```



```
"Result: 16"
```

Expressions

Variables and literals are interchangeable! Both of these do the same thing:

```
name = "Bison"
opening = "Hello, "
greeting = opening + name
print(greeting)
```

```
name = "Bison"
opening = "Hello, "
print(opening + name)
```

Expressions

What is the expression here?

```
name = "Bison"
opening = "Hello, "
greeting = opening + name
print(greeting)
```

```
name = "Bison"
opening = "Hello, "
print(opening + name)
```

Expressions

What is the expression here?

```
name = "Bison"
opening = "Hello, "
greeting = opening + name
print(greeting)
```

```
name = "Bison"
opening = "Hello, "
print(opening + name)
```

Code works fine, whether we assign the expression value to a name or not.

Input

Input

We learned “output” for how to print stuff out to the terminal. Now, we will learn how to take things in from the terminal.

New function!

```
answer = input("Prompt")
```

Input

We learned “output” for how to print stuff out to the terminal. Now, we will learn how to take things in from the terminal.

New function!

```
answer = input("Prompt")
```

Prints “Prompt” to the screen, then waits for user to provide input.

Input

We learned “output” for how to print stuff out to the terminal. Now, we will learn how to take things in from the terminal.

New function!

```
answer = input("Prompt")
```

Prints “Prompt” to the screen, then waits for user to provide input.

We **assign** the words that the user types in to the variable *answer*.