

DS5010

Intro to programming for Data Science

LECTURE 1

TODAY

- syllabus overview
- house and alias name assignments
- basic python objects
- basic operations
- variables and types
- rules of sum and product
- branching and conditionals

Syllabus

□ you can access the full syllabus here:

<https://rahiminasab.github.io/DS5010F19/>

House and Alias name Assignments

- students in this class will be divided into 5 Houses:
 - Stark
 - Baratheon
 - Lannister
 - Targaryen
 - Martell
- each student will also get an alias name which are chosen from GOT character names.
- students will participate in HackerRank contests with a username derived from their alias names.
- a script written in Python will do the assignments and emails the students their assigned, House, alias name, and username.

Python vs English as a language

primitive constructs

- English: words.
- programming language: numbers, strings, simple operators.



Word Cloud copyright [Michael Twardos](#), All Right Reserved.

```
float **
* <= < bool
string >= !=
int /
NoneType —
```

Python vs English as a language

□ syntax

- English: **"cat dog boy"** → not syntactically valid
"cat hugs boy" → syntactically valid
- programming language: **+"hi"5** → not syntactically valid
3.2*5 → syntactically valid

Python vs English as a language

□ **static semantics** is which syntactically valid strings have meaning

- English: **"I are hungry"** → syntactically valid but with static semantic error
- programming language: **3.2*5** → syntactically valid
3+"hi" → static semantic error

Python vs English as a language

□ **semantics** is the meaning associated with a syntactically correct string of symbols with no static semantic errors

- English: can have many meanings "Flying planes can be dangerous"
- programming languages: have only one meaning but may not be what programmer intended

WHERE THINGS GO WRONG

❑ **syntactic errors**

- common and easily caught

❑ **static semantic errors**

- some languages check for these before running program
- can cause unpredictable behavior

❑ no semantic errors but **different meaning than what programmer intended**

- program crashes, stops running
- program runs forever
- program gives an answer but different than expected

PYTHON PROGRAMS

- ❑ a **program** is a sequence of definitions and commands
 - definitions **evaluated**
 - commands **executed** by Python interpreter in a shell
- ❑ **commands**(statements) instruct interpreter to do something
- ❑ can be typed directly in a **shell** or stored in a **file** that is read into the shell and evaluated

OBJECTS

- ❑ programs manipulate **data objects**
- ❑ objects have a **type** that defines the kinds of things programs can do to them
 - ❑ A human can walk, and speak English
 - ❑ A monkey can walk, but cannot speak English
- ❑ objects are
 - ❑ scalar (cannot be subdivided)
 - ❑ non-scalar (have internal structure that can be accessed)

SCALAR OBJECTS

- ❑ int – represent **integers**, ex. 5
- ❑ float – represent **real numbers**, ex. 3.27
- ❑ bool – represent **Boolean** values True and False
- ❑ NoneType – **special** and has one value, None
- ❑ can use type() to see the type of an object

```
>>> type(5)
```

```
int
```

```
>>> type(3.0)
```

```
float
```

*what you write into
the Python shell*

*what shows after
hitting enter*

TYPE CONVERSIONS (CAST)

- ❑ can **convert object of one type to another**
- ❑ `float(3)` converts integer 3 to float 3.0
- ❑ `int(3.9)` truncates float 3.9 to integer 3

PRINTING TO CONSOLE

- to show output from code to a user, use print command

```
>>> 3+2
```

```
In [11]: 3+2
```

```
Out[11]: 5
```

*"Out" tells you it's an
interaction within the
shell only*

```
In [12]: print(3+2)
```

```
5
```


*No "Out" means it is
actually shown to a user,
apparent when you
edit/run files*

EXPRESSIONS

- ❑ **combine objects and operators** to form expressions
- ❑ an expression has a **value**, which has a type
- ❑ syntax for a simple expression

`<object> <operator> <object>`

OPERATORS ON ints and floats

- $i+j$ → the sum
 - $i-j$ → the difference
 - $i*j$ → the **product**
 - i/j → **division**
 - $i//j$ → **floor division**
- if both are ints, result is int
if either or both are floats, result is float
- result is float
- $i\%j$ → the **remainder** when i is divided by j
 - $i**j$ → i to the **power** of j
- 

SIMPLE OPERATIONS

- parentheses used to tell Python to do these operations first
- **operator precedence** without parentheses
 - **
 - *
 - /
 - + and – executed left to right, as appear in expression

BINDING VARIABLES AND VALUES

- equal sign is an **assignment** of a value to a variable name

variable
`pi` = `3.14159`
value

`pi_approx = 22/7`

- value stored in computer memory
- an assignment binds name to value
- retrieve value associated with name or variable by invoking the name, by typing `pi`

ABSTRACTING EXPRESSIONS

- why **give names** to values of expressions?
- to **reuse names** instead of values
- easier to change code later

```
pi = 3.14159  
radius = 2.2  
area = pi*(radius**2)
```

PROGRAMMING vs MATH

- in programming, you do not “solve for x”

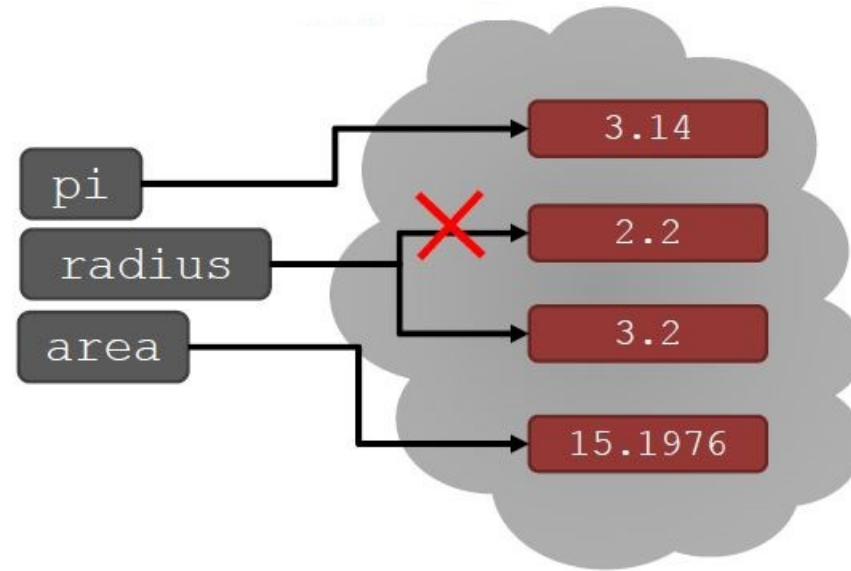
```
pi = 3.14159
radius = 2.2
# area of circle
area = pi*(radius**2)
radius = radius+1
```

an assignment
* expression on the right, evaluated to a value
* variable name on the left
* equivalent expression to `radius = radius + 1`
is `radius += 1`

CHANGING BINDINGS

- can **re-bind** variable names using new assignment statements
- previous value may still stored in memory but lost the handle for it
- 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)  
radius = radius+1
```



A SIMPLE COUNTING PROBLEM

- You know **4** different roads which you can take to go from NEU to Copley square
- You know **3** different roads which you can take to go from Copley to Downtown.
- ❖ In **how many different ways** can you go from NEU to Downtown given that you want to stop at Copley in between?
- You know also **5** different roads which you can take to go from NEU to Downtown directly without passing Copley.
- ❖ In how many different ways can you go from NEU to Downtown now?

DM: RULES OF SUM and PRODUCT

Rule of sum:

- if we have A ways of doing something and B ways of doing another thing and we can not do both at the same time, then there are $A + B$ ways to choose one of the actions.

Rule of product:

- if there are a ways of doing something and b ways of doing another thing, then there are $a * b$ ways of performing both actions.

STRINGS

- letters, special characters, spaces, digits
- enclose in **quotation marks or single quotes**

```
hi = "hello there"
```

- **concatenate** strings

```
name = "Arya"
```

```
greet = hi + name
```

```
greeting = hi + " " + name
```

- do some **operations** on a string as defined in Python docs

```
silly = hi + " " + name * 3
```


INPUT/OUTPUT: print

- used to **output** stuff to console
- keyword is `print`

```
x = 1
```

```
print(x)
```

```
x_str = str(x)
```

```
print("my fav num is", x, ".", "x =", x)
```

```
print("my fav num is " + x_str + ". " + "x = " + x_str)
```

INPUT/OUTPUT: input("")

- prints whatever is in the quotes
- user types in something and hits enter
- binds that value to a variable

```
text = input("Type anything... ")  
print(5*text)
```

- input **gives you a string** so must cast if working with numbers

```
num= int(input("Type a number... "))  
print(5*num)
```

STRINGS

- You can access the *i*'th character in a string using brackets.

```
s = "abcde"
```

```
s[0] → 'a'
```

```
s[1] → 'b'
```

```
...
```

```
s[4] → 'e'
```

- use `len` function to get the length of a string, which is the number of characters it has:

```
len(s) → 5
```

- Negative indexing, makes it easy to access the last elements!

```
s[-1] → 'e'
```

STRINGS

- you can get a slice of a string, by telling that from which index it starts and before which one it ends.

```
s = "abcdef"
```

```
s[1:4] → 'bcd'
```

```
s[:3] → 'abc'   if we do not write the starting index, it assumes 0
```

```
s[3:] → 'def'   if we do not write the ending index, it assumes it is len(s)
```

COMPARISON OPERATORS ON int, float, string

- `i` and `j` are variable names
- comparisons below evaluate to a Boolean

`i > j`

`i >= j`

`i < j`

`i <= j`

`i == j` → **equality** test, True if `i` is the same as `j`

`i != j` → **inequality** test, True if `i` not the same as `j`

LOGIC OPERATORS ON bools

- `a` and `b` are variable names (with Boolean values)

not a → True if `a` is False. False if `a` is True

a and b → True if both are True

a or b → True if either or both are True

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

COMPARISON EXAMPLE

```
pset_time = 15  
sleep_time = 8  
print(sleep_time > pset_time)  
derive = True  
drink = False  
both = drink and derive  
print(both)
```

CONTROL FLOW - BRANCHING

```
if <condition>:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
elif <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

- <condition> has a value True or False
- evaluate expressions in that block if <condition> is True

INDENTATION

- matters in Python
- how you denote blocks of code

```
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))

if x == y:
    print("x and y are equal")
    if y != 0:
        print("therefore, x / y is", x/y)
elif x < y:
    print("x is smaller")
else:
    print("y is smaller")

print("thanks!")
```