Programming for Analytics

Intro to Python, data types, statements, I/O

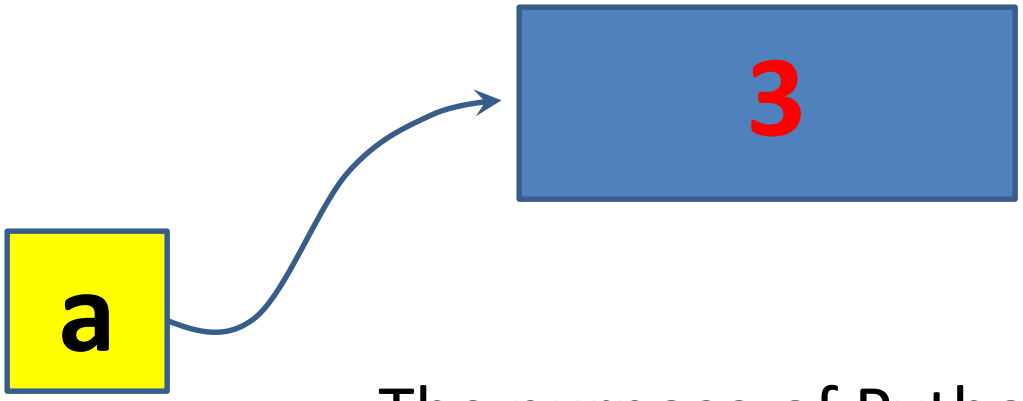
S. Kanungo

Numeric variables

- Four types
  - integer, float, long, complex
- For every variable type there is a defined set of operators
- assignment is the most common operator
  - Aka assignment statement
  - `<variable> = <expression>`

Assignment statement

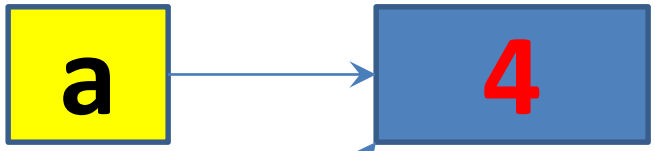
a = 3



The purpose of Python's assignment statement is to associate **names** with **values** in your program.

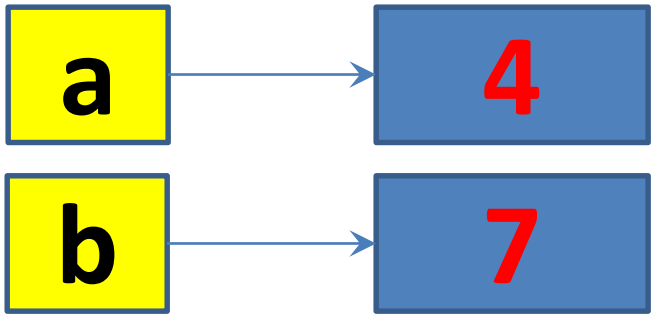
Understanding assignment

```
>>> a = 4
```




```
>>> b = a
```


```
>>> b = 7
```



Assignment Demo

- Assign 3 to identifier `myint`
- Is `myint` a variable? Why?
- Assign 4.3 to identifier `myreal`
- Assign 3.3 to `myint`
- What happened here?

Standard operations on numerics - 1

Operation	Result
x + y	sum of *x* and *y*
x - y	difference of *x* and *y*
x * y	product of *x* and *y*
x / y	quotient of *x* and *y*
x // y	(floored) quotient of *x* and *y*
x % y	remainder of x / y
-x	*x* negated
+x	*x* unchanged

Reference: https://docs.python.org/3/library/stdtypes.html

Note the differences between Python 2 and Python 3:
http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html#integer-division

Standard operations on numerics - 2

Operation	Result
abs(x)	absolute value or magnitude of *x*
int(x)	*x* converted to integer
long(x)	*x* converted to long integer
float(x)	*x* converted to floating point
complex(re, im)	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.
c.conjugate()	conjugate of the complex number *c*. (Identity on real numbers)
divmod(x, y)	the pair (x // y, x % y)
pow(x, y)	*x* to the power *y*
x ** y	*x* to the power *y*

Reference: https://docs.python.org/3/library/stdtypes.html

Operators Demo

- Do this using the IDLE interface or Jupyter

```
v1 = 6
v2 = 3.5
v1 / v2
v1 // v2
v1 % v2
v3 = True
v1 < v2
v1 + v6 - v7
v1 = None
del (var2)
```

- `dir()` will give you the list of in scope variables:
- `globals()` will give you a dictionary of global variables
- `locals()` will give you a dictionary of local variables

More data types

- Two more data types
  - None
  - Boolean
    - Operations always return 0 (False) or 1 (True)
    - Boolean operations, ordered by ascending priority

Operation	Result
x or y	if *x* is false, then *y*, else *x*
x and y	if *x* is false, then *x*, else *y*
not x	if *x* is false, then True, else False

- What does the **`del`** command do?

Sequence Types

- **str:** self explanatory; string literals are written in single or double quotes
- **unicode**: Unicode strings are much like strings, but are specified in the syntax using a preceding 'u' character: u'abc', u"def".
- **list**: Lists are constructed with square brackets, separating items with commas: [a, b, c].
- **tuple**: A tuple consists of a number of values separated by commas
- **bytearray**: mutable sequence of integers in the range 0 <= x < 256
- **buffer**: objects are not directly supported by Python
- **xrange**: similar to buffers in that there is no specific syntax to create them

Strings

- A string is an immutable object
- A string is a sequence of characters
- There are four ways to quote string literals:

  - Using ' quote, e.g., `'A string literal'`
  - Using " quote, e.g., `"A string literal"`
  - Using """ quote for multi-line strings, e.g.,

    ```
    """A string which contains
    more than one line"""
    ```

    Line breaks are maintained in the string as newline characters.
  - Raw strings don't process \ escapes, mainly used with regular expressions

    ```
    r"This is a raw string"
    ```

Strings

- Other than raw strings, special characters can be included by using backslash escape codes, e.g,

```
"Here\tis a tab character and this\n is a newline"
```

Some of the common escape characters include

\n	Newline (linefeed)
\t	horizontal tab
\\	a single backslash
\"	a double-quote
\'	a single-quote (apostrophe)
\e	an ASCII escape character

- Adjacent string literals are concatenated

```
"This is just"  " one long string"
```

Strings

- A string is an immutable object
- A string is a sequence of characters
- There are four ways to quote string literals:

  - Using ' quote, e.g., `'A string literal'`
  - Using " quote, e.g., `"A string literal"`
  - Using """ quote for multi-line strings, e.g.,

    ```
    """A string which contains
    more than one line"""
    ```

    Line breaks are maintained in the string as newline characters.
  - Raw strings don't process \ escapes, mainly used with regular expressions

    ```
    r"This is a raw string"
    ```

Strings

- Other than raw strings, special characters can be included by using backslash escape codes, e.g,

```
"Here\tis a tab character and this\n is a newline"
```

Some of the common escape characters include

\n	Newline (linefeed)
\t	horizontal tab
\\	a single backslash
\"	a double-quote
\'	a single-quote (apostrophe)
\e	an ASCII escape character

- Adjacent string literals are concatenated

```
"This is just"  " one long string"
```

Understanding raw strings

- Try this

```
print("Here\tis a tab character and this\n is a newline")
print (r"Here\tis a tab character and this\n is a newline")
a = "Here\tis a tab character and this\n is a newline"
print(a)
print (repr(a))
```

- The `repr` function is often used to obtain a printable representation of the object, in our case the string

String Demo

- Do this using the IDLE interface or Jupyter

```
s1 = "Programming for Analytics"
print s1
```

```
print (s1[0])
print (s1[-1])
print (s1[-2])
print (s1[2:6])
print (s1[4:])
print (s1[:10])
```

```
print ("DN" in s1)
print ("for" in s1)
```

```
print (s1.title())
print (s1.swapcase())
print (s1.upper())
print (s1.lower())
print (s1.capitalize())
```

```
print (s1.count('a'))
print (s1.find('mm'))
print (s1[s1.index('mm'):])
print (s1.rfind('a'))
print (s1[s1.index('a'):s1.rindex('a')])
s2 = s1.replace('a', 'Z')
print (s2)
```

Lists – some operators

- Unlike strings, lists are mutable.
- When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.
- We explore lists by
  - Defining
  - Adding Elements
  - Searching
  - Deleting Elements
  - Using Operators

```
mylist = s1.split(' ')
print (mylist)
s3 = mylist[1] + ' ' + mylist[2]
print (s3)
```

List Demo

- Do this using the IDLE interface or Jupyter

```
s4 = 'a3a'
s[1]
s[1] = 'a'  # Error, why?
```

```
ls5 = []
ls5.append('a')
ls5.append('3')
ls5.append('a')
ls
```

```
ls5[1] = ('3')
ls
```

1. Create a list of four of your friends
2. Add one more friend to the list
3. Search for a specific friend in your list
4. Delete the second friend in your list
5. Use the “+” operator on your list

Built-in functions

- Pre-programmed functions are available in Python
  https://docs.python.org/3/library/functions.html

		Built-in Functions	
abs()	divmod()	input()	ope
all()	enumerate()	int()	ord
any()	eval()	isinstance()	pow
basestring()	execfile()	issubclass()	pri

- Do get used to searching the official Python documentation

Built-in function exercise

- This is to be completed using the IDLE interface

Use the following functions

```
range()
pow()
sum()
ord()
sorted()
format()
```

Modules in Python

- A module in Python is just a file containing Python definitions and statements.
- Modules allow for modular programming.
- Built-in modules and user-defined modules exist.
- More on this later

Boolean operations and constants

- Boolean operations evaluate to **True** or **False**
- **True** and **False** are constants in Python.
- Equivalents of **True** and **False** exist
  - **False**: zero, None, empty container or object
  - **True**: non-zero numbers, non-empty objects
- Comparison operators
  - ==, !=, <>, <, <=, >, >
- Example
  - 2 == 2 evaluates to **True**

Boolean logic

- Combinations of Boolean expressions are very useful
- The following expressions evaluate to ...
  - if a is True and b is True: **a and b** **True**
  - if a is True or b is True: **a or b** **True**
  - if a is False: **not a** **True**
- Employ parentheses as needed disambiguate complex Boolean expressions.

Interesting behavior: `and` and `or`

- **`and`** and **`or`** don't return **`True`** or **`False`**
  - They return the value of one of their sub-expressions, which may be a non-Boolean value
    - `X and Y and Z`
      - If all are true, returns value of Z
      - Otherwise, returns value of first false sub-expression
    - `X or Y or Z`
      - If all are false, returns value of Z
      - Otherwise, returns value of first true sub-expression
- **`and`** and **`or`** use *lazy evaluation*, so no further expressions are evaluated

Conditional expressions

- x = true_value if condition else false_value
- Uses lazy evaluation:
  - First, condition is evaluated
  - If *True*, true_value is evaluated and returned
  - If *False*, false_value is evaluated and returned
- Standard use
  - x = (true_value if condition else false_value)

`Try/except` statements

- Errors detected during execution are called *exceptions* and are not unconditionally fatal
- They can be handled using the `try` / `except` block. It looks like the following:

```
try:
   something
except:
   handle the exception
```

Example of `try/except`

```
(x,y) = (5,0)
try:
  z = x/y
except ZeroDivisionError:
  print "divide by zero"
```

Specific

```
(x,y) = (5,0)
try:
  z = x/y
except ZeroDivisionError as e:
  z = e
  print z
```

Broader

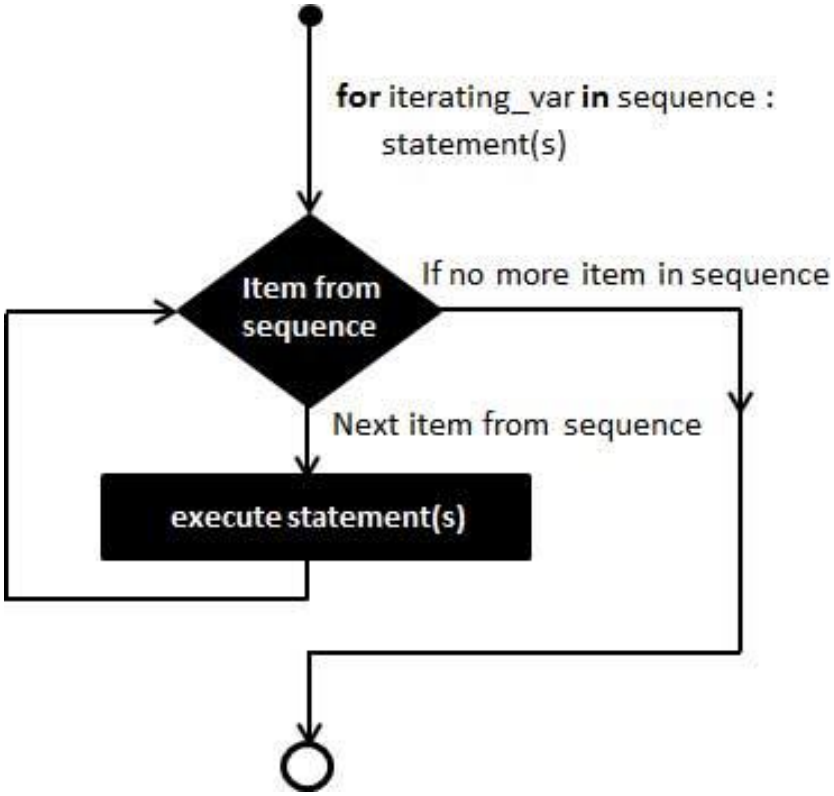
```
(x,y) = (5,0)
try:
  z = x/y
except Exception as e:
  print str(e)
  print repr(e)
```

Even broader
this will print the exception
this will print name of exception and value

For loop

- In general, used when you know how many repetitions you want

```
for i in range(5):
    print (i)
    print (i**2)
    print (i*i*i)
```



Try this

- In general, used when you know how many repetitions you want
- You will need to write a for loop to average some numbers
- Ask the user how many numbers s/he wants to average.
- Loop the number of times the user inputs and request the user for a numeric input
- Finally compute the average and print it out

range

- `range` function creates a list containing numbers defined by the input.

```
for x in range(0, 4, 2):
    print ("This is index %d" % (x))

for x in range(7, 0, -3):
    print ("This is index %d" % (x))
```

range (cont.)

- The most common way of using range is without specifying the “step”
- Also, you need to know how to break out of a loop

```
for x in range(0, 3):
    for y in xrange(0, 2):
        print ('%d * %d = %d' % (x, y, x*y))
```

```
for x in range(300000):
    print (x)
    if x == 11:
        break
```

Strings and lists as iterables

- You can iterate almost over anything in python using the `for` loop construct
- E.g., we can iterate over a string or over a list

```
mystring = "Hello there"
for x in mystring:
    print (x)
```

```
mycollection = ['Hello', True, 5, 5.0, (2,3)]
for x in mycollection:
    print (x)
```

While loops

- When we can't know how many iterations we need we typically end up using the while statement or while loop

```
i = 0
while i <= 10:
    print(i)
    i = i + 1
```

```
for i in range(11):
    print(i)
```

- Typically used to implement interactive and sentinel loops

Try this

- Generate a random number between 0 and 100 and give the user 6 chances to guess the number.
- Every time the user guesses the wrong number, the message to the user should be whether the guess is too high or too low.
- The program exits when the user guesses the number correctly or when the six chances are used up.

Try this (optional)

Write a program that presents the following menu to the user

1. Add two numbers
2. Subtract two numbers
3. Quit

The program should continue running (showing this menu) until the user enters 3 and presses enter. For choices 1 and 2, the program should request the user for two numeric inputs, validate that they are numeric, keep prompting the user till the user provides valid numeric inputs or the user types in “C” or “c” (to cancel) for either of the inputs. If the user provides valid numeric inputs your output should look like “The sum of 5.2 and 2.33 is 7.53” or “The difference of 4.2 and 2.33 is 1.87”. If the user chooses to cancel, the menu is shown again.

Nested loops

- Assume that we have a list of list of integers
- Further assume that each list can be a different size
- Use for loop(s) to print all numbers (one number per line, as shown alongside

```
List 1, Number  1: 4
List 1, Number  2: 3
List 1, Number  3: 10
List 2, Number  1: 1
List 2, Number  2: 12
List 3, Number  1: 7
List 3, Number  2: 7
List 3, Number  3: 2
List 3, Number  4: 6
List 3, Number  5: 11
List 4, Number  1: 7
…
…
```

The `break` loop control

- You have already seen this in these slides
- The break statement in Python terminates the current loop and resumes execution at the next statement

```
# First Example
for letter in 'Python':
if letter == 'h':
      break
   print ('Current Letter :', letter)
```

```
# Second Example
var = 10
while var > 0:
   print ('Current variable value :', var)
   var = var -1
   if var == 5:
      break
print ("Good bye!“)
```

The `continue` loop control

- The continue statement
  - rejects all the remaining statements in the current iteration of the loop and
  - moves the control back to the top of the loop.

```
# First Example
for letter in 'Python':
   if letter == 'h':
      continue
   print ('Current Letter :', letter)
```

```
# Second Example
var = 10
while var > 0:
   var = var -1
   if var == 5:
      continue
   print ('Current variable value :', var)
print ("Good bye!“)
```

The `for … else` loop control

- When would you use `for … else`
- Alternatives to this idiom

```
for i in some_iterable:
  if some_iterable(i) == some_condition:
    break
else:
  do_something()
```

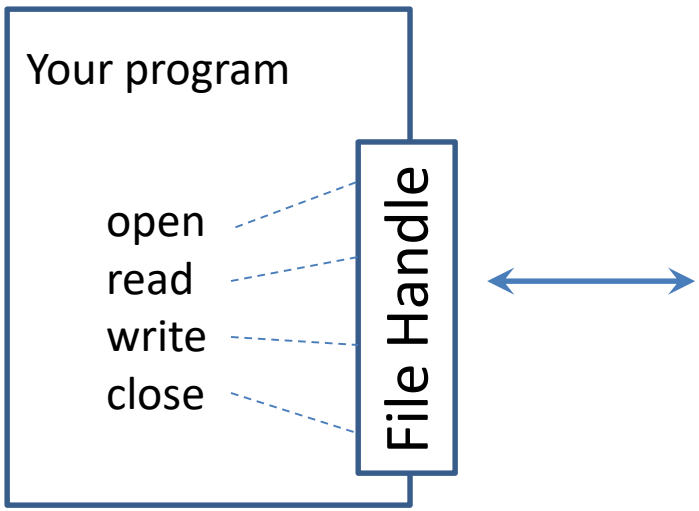
- the `else` suite is executed after the `for`, but only if the `for` terminates *normally* (not by a `break`).

`for … else` exercise

- Start with `mylist = [1,2,4,3,5,6]`
- Loop through this list; Stop as soon as you find an even number and print "list contains an even number"
- If you traverse the entire list without finding an even number, print "list does not contain an even number"

Accessing files in Python

- Files typically reside in non-volatile memory
- Require a file handle



Frost.txt

```
Stopping by Woods on a Snowy Evening
BY ROBERT FROST

Whose woods these are I think I know.
His house is in the village though;
He will not see me stopping here
To watch his woods fill up with snow.

My little horse must think it queer
To stop without a farmhouse near
Between the woods and frozen lake
The darkest evening of the year.
…
…
…
```

Reading text files

- A typical workflow when we read files
  - Identify the file name and path
  - Obtain a handle to read that file
  - Read that file and do whatever else you want
  - Return the file handle

Preferred usage

- Use of the **`with`** statement
- Why
  - Too many opens without closes
  - Exceptions my occur

```
with open('my_file', 'r') as f:
    for line in f:
        # do stuff
```

Three types of *reads*

- `read(size)`
  - Reads characters
- `readline()`
  - Reads line
- `readlines()`
  - Reads all lines

Writing to text files

- Once again, we will work our way through a typical workflow
  - Identify the file name and path
  - Obtain a handle to write to that file
  - Perform the write(s) and whatever else
  - Return the file handle

Class problem 1

Compute the average of numbers that a user enters interactively. There is no need to do validation (assume that valid input is provided in the form of a number). Read everything as a float. Instruct the user that once s/he is done with providing numbers, the last number should be a negative number Finally print the average of the list of numbers.

Class problem 2

Assume that we wanted to add numbers stored in a file. The file contains the following data:

```
12,23,12,14,15
1,34,2,21
2,4,7,6,9,12,34,3
23,45
1,23,87
```

The file is named **numbers.txt**
Your task is to read the file and provide the following output:

```
There are xx numbers in the file.
The sum of all the numbers is xx.
The average if all the numbers is xx.
```

Python string formatting

```
bags = 34
stuff = "Oatmeal"
weight = 123.3467

# Old C-style formatting in Python
print "We have %d bags of %s weighing %4.3f
lbs." % (bags, stuff, weight)

# Python-style formatting
print "We have {:d} bags of {} weighing
{:.3f} lbs.".format(bags, stuff, weight)
```

Old c-style format codes

%s	Format a string. For example, '%-3s' % 'xy' yields 'xy '; the width (-3) forces left alignment.
%d	Decimal conversion. For example, '%3d' % -4 yields the string ' -4'.
%e	Exponential format; allow four characters for the exponent. Examples: '%08.1e' % 1.9783 yields '0002.0e+00'.
%E	Same as %e, but the exponent is shown as an uppercase E.
%f	For float type. E.g., '%4.1f' % 1.9783 yields ' 2.0'.
%g	General numeric format. Use %f if it fits, otherwise use %e.
%G	Same as %G, but an uppercase E is used for the exponent if there is one.
%o	Octal (base 8). For example, '%o' % 13 yields '15'.
%x	Hexadecimal (base 16). For example, '%x' % 247 yields 'f7'.
%X	Same as %x, but capital letters are used for the digits A-F. For example, '%04X' % 247 yields '00F7'; the leading zero in the length (04) requests that Python fill up any empty leading positions with zeroes.
%c	Convert an integer to the character with the corresponding ASCII code. For example, '%c' % 0x61 yields the string'a'.
%%	Places a percent sign (%) in the result. Does not require a corresponding value. Example: "Energy at %d%%." % 88yields the value 'Energy at 88%.'.

Python string formatter

Number	Format	Output	Description
3.1415926	{:.2f}	3.14	2 decimal places
3.1415926	{:+.2f}	+3.14	2 decimal places with sign
-1	{:+.2f}	-1.00	2 decimal places with sign
2.71828	{:.0f}	3	No decimal places
5	{:0>2d}	05	Pad number with zeros (left padding, width 2)
5	{:x<4d}	5xxx	Pad number with x's (right padding, width 4)
10	{:x<4d}	10xx	Pad number with x's (right padding, width 4)
1000000	{:,}	1,000,000	Number format with comma separator
0.25	{:.2%}	25.00%	Format percentage
1000000000	{:.2e}	1.00e+09	Exponent notation
13	{:10d}	13	Right aligned (default, width 10)
13	{:<10d}	13	Left aligned (width 10)
13	{:^10d}	13	Center aligned (width 10)