This PPT is for GitHub. All materials are for educational purposes.

COM5507 Social Media Data Acquisition and Processing
Week 2 – 3. Python in Action: A Command Liner

Lectured by: Dr. Xinzhi ZHANG
Research Assistant Professor, Department of Journalism
Hong Kong Baptist University
12 Sep 2018 @ CityU M5064

Agenda

- Computational thinking
- Python as a computer programming language
  - Program execution
  - Program steps and flow
  - Errors and debugging
- Vocabulary/Words
  - Variables & expressions
- Data Structures
- Functions
- Sentences & Paragraphs
  - Control flow statements
  - Conditions
  - Loops and iterators

Some references

- A command-liner: https://www.py4e.com/book.php
- Check the codes and develop the computational thinking: http://interactivepython.org/runestone/static/thinkcspy/index.html
- A good and concise tutorial based on Jupyter Notebook (check the codes): https://github.com/jakevdp/WhirlwindTourOfPython
- Another option to learn a programming language in general: *Learn Python the hard way* (but not covered in this course)

The Python Tutorial

- https://docs.python.org/3.7/tutorial/index.html

“Computational thinking”

- To think like a computer scientist = mathematics, engineering, and natural science.
- *Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations).*
- *Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives.*
- *Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.*
- *The single most important skill for a computer scientist is problem solving. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately.*

- -- Allen B. Downey, *How to think like a computer scientist: Learning with Python*

Guido van Rossum

Guido van Rossum is a Dutch programmer best known as the author of the Python programming language, for which he was the "Benevolent Dictator For Life" (BDFL) until he stepped down from the position in July 2018.

Tips to learn a programming language

- Read and write
- Pay attention to the details
- Discover and compare the differences

- *--- Zed Shaw: learn python the hard way*

Executing (“running”) the program

- This week we will be a “command liner.”
- (Next week we shall move to the Jupyter Notebook).

- Executing your first Python program:
- *01_print.py*

Interpreter and compiler

- CPU only understands machine language (11000101100101010…….)
- An interpreter reads the source code of the program as written by the programmer, parses the source code, and interprets the instructions.
- It is the nature of an interpreter to be able to have an interactive conversation.
- A compiler needs to be handed the entire program in a file, and then it runs a process to translate the high-level source code into machine language and then the compiler puts the resulting machine language into a file for later execution.
- ".exe" or ".dll" which stand for "executable" and "dynamic link library" respectively. In Linux and Macintosh, there is no suffix that uniquely marks a file as executable though.

Program steps or program flow

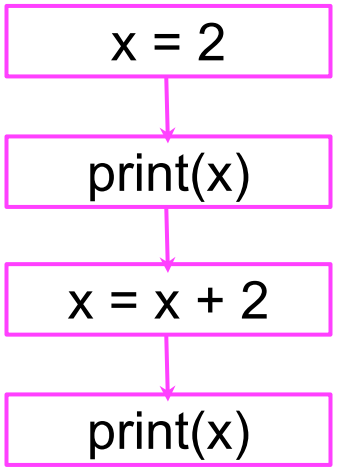
- **input**: Get data from the "outside world". This might be reading data from a file, or even some kind of sensor like a microphone or GPS. In our initial programs, our input will come from the user typing data on the keyboard.
- **output**: Display the results of the program on a screen or store them in a file or perhaps write them to a device like a speaker to play music or speak text.
- **sequential execution**: Perform statements one after another in the order they are encountered in the script.
- **conditional execution**: Check for certain conditions and then execute or ***skip*** a sequence of statements.
- **repeated execution**: Perform some set of statements repeatedly, usually with some variation.
- **reuse**: Write a set of instructions once and give them a name and then reuse those instructions as needed throughout your program.

Program flow: input

- We can instruct Python to pause and read data from the user using the input() function
- The input() function returns a string
- We can also input a file (more on file inputs later)

- *02a_input.py*
- *02b_count.py*
- *03_countall.py*

Program flow: Sequential steps



Program:

```
x = 2
print(x)
x = x + 2
print(x)
```

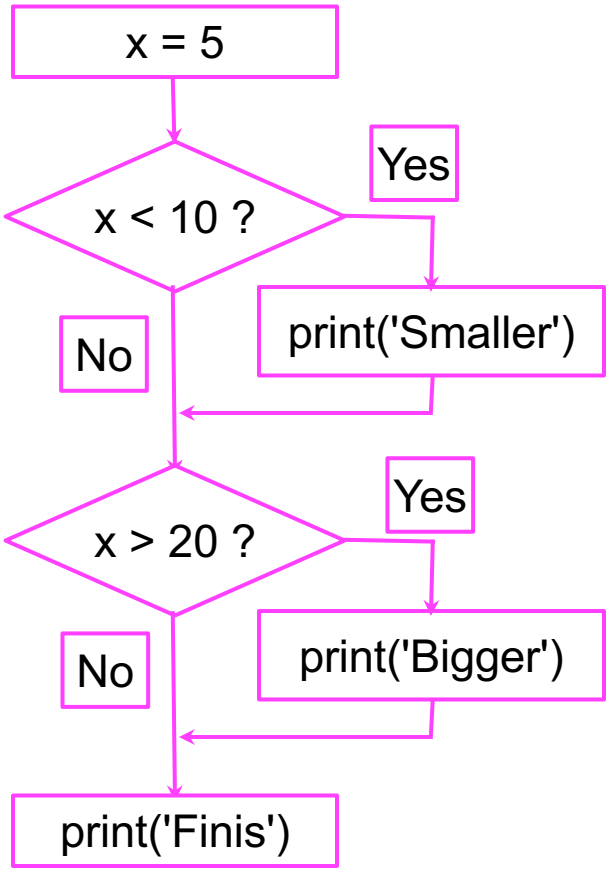
Output:

```
2
4
```

When a program is running, it flows from one step to the next. As programmers, we set up "paths" for the program to follow.

Source: https://www.py4e.com/html3/

Program flow: Conditional steps



Program:

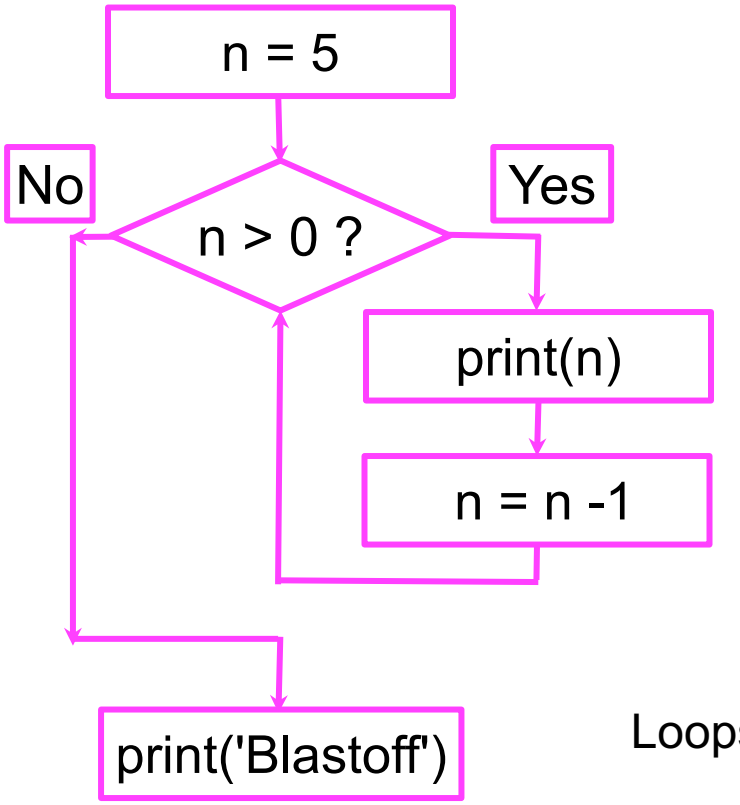
```
x = 5
if x < 10:
    print('Smaller')
if x > 20:
    print('Bigger')

print('Finish')
```

Output:

```
Smaller
Finis
```

Source: https://www.py4e.com/html3/

Program flow: Repeated steps



Program:

```
n = 5
while n > 0 :
    print(n)
    n = n – 1
print('Blastoff!')
```

Output:

5
4
3
2
1
Blastoff!

Loops (repeated steps) have iteration variables that change each time through a loop.

Source: https://www.py4e.com/html3/

Errors and debugging

- Programming errors are called *bugs* and the process of tracking them down and correcting them is called debugging.
- These kinds of errors can occur in a program:
  - syntax errors,
  - runtime errors,
  - semantic errors,
  - There might be other errors

Errors and debugging

- Syntax errors. These are the first errors you will make and the easiest to fix. A syntax error means that you have violated the "grammar" rules of Python. The line and character that Python indicates in a syntax error may just be a starting point for your investigation.
- Runtime error. The error does not appear until you run the program. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened.
- Semantic errors. A semantic error is when your description of the steps to take is syntactically perfect and in the right order, but there is simply a mistake in the program. The program is perfectly correct but it does not do what you intended for it to do.
- Debugging
  - Test (more will be covered in later sections)
  - Google the error message.
  - “Asking the reproducible questions” (week 1) on the Forum

Variables and expressions

- Constants
  - fixed values. Can be both numeric or string (with a quote)
- Variables: a named place in the memory where a programmer can store data and later retrieve the data using the variable “name”
  - Programmers get to choose the names of the variables
  - You can change the contents of a variable in a later statement
- Assignments: In Python, we assign a value to a variable using the assignment statement (=)

How to name a variable?

- Must start with a letter or underscore _
- Must consist of letters, numbers, and underscores
- Case Sensitive
- Cannot use reserved words

Reserved words

- You cannot use reserved words as variable names / identifiers

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

Numeric expressions

- Because of the lack of mathematical symbols on computer keyboards - we use “computer-speak” to express the classic math operations
- Asterisk is multiplication
- Exponentiation (raise to a power) looks different than in math.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
%	Remainder

- *04_var.py*

Operator precedence rules

- Highest precedence rule to lowest precedence rule:
  - Parentheses are always respected
  - Exponentiation (raise to a power)
  - Multiplication, Division, and Remainder
  - Addition and Subtraction
  - Left to right

Data structures in Python

- All Python objects have ***type information*** attached.
- Built-in data types
- Built-in data structures. A data structure is a particular way of organizing data in a computer.

“Simple” data types

Type	Example	Description
int	x = 1	integers (i.e., whole numbers)
float	x = 1.0	floating-point numbers (i.e., real numbers)
complex	x = 1 + 2j	Complex numbers (i.e., numbers with real and imaginary part)
bool	x = True	Boolean: True/False values
str	X = “abc”	String: characters or text. A string is a sequence of characters.
NoneType	X = None	Special object indicating nulls

This table is revised and modified from:
https://github.com/jakevdp/WhirlwindTourOfPython/blob/master/05-Built-in-Scalar-Types.ipynb

“Simple” data structures in Python

Type	Example	Description	Can we change the content within this collection?	Are the items in the collection ordered?
list	[1, 2, 3]	List: an ***mutable***, ***ordered*** collection	Yes – mutable	Yes – ordered
tuple	(1, 2, 3)	Tuple: an ***immutable, ordered*** collection	No – immutable	Yes – ordered
set	{1, 2, 3}	Set: ***Unordered*** collection of unique values	Yes – mutable	No – unordered
dict	{'a':1, 'b':2, 'c':3}	Dictionary: ***Unordered*** (key, value) mapping	Yes – mutable	No – unordered

Table compiled by Xinzhi Zhang on 20180912

Lists

- A list is a kind of collection, a sequence of values.
- Lists: integers, strings, empty, and nested lists
- Lists are mutable
- Checking the list
  - how many elements?
  - something in the list?
- Working on lists
  - traversing
  - indexing (zero-based, and -1 from right to left)
  - concatenating and slicing

- *05_ds_list_1.py*
- *05_ds_list_2.py*

Tuples

- Tuples are in many ways similar to lists, but they are defined with parentheses rather than square brackets.
- Both lists and tuples have elements which are indexed starting at 0.
- Tuples are immutable.
- Tuples are efficient in making some temporary variables, especially if we have multiple value outputs.

- *05_ds_tuple.py*

Dictionaries

- Dictionaries are flexible mappings of keys to values, and form the basis of much of Python's internal implementation.
- They can be created via a comma-separated list of ***key:value pairs*** within curly braces.

- *05_ds_dic*

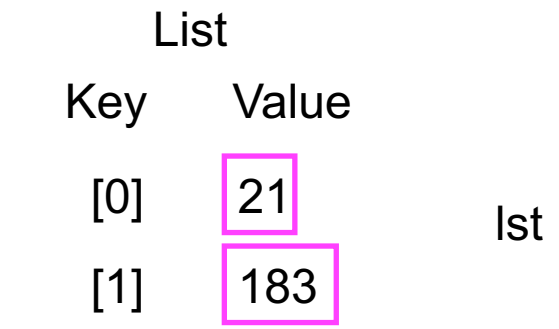
Lists vs dictionaries

- Dictionaries are like lists except that they use keys instead of numbers to look up values

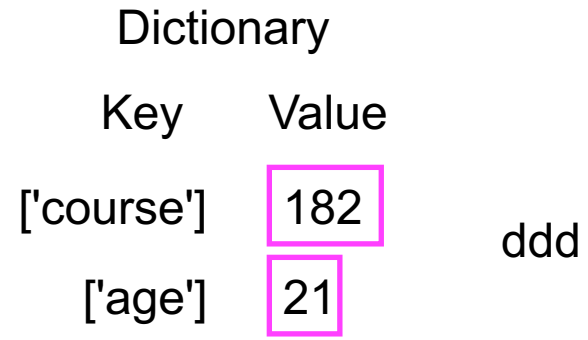
```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print(lst)
[21, 183]
>>> lst[0] = 23
>>> print(lst)
[23, 183]
```

```
>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 182
>>> print(ddd)
{'course': 182, 'age': 21}
>>> ddd['age'] = 23
>>> print(ddd)
{'course': 182, 'age': 23}
```

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print(lst)
[21, 183]
>>> lst[0] = 23
>>> print(lst)
[23, 183]
```




```
>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 182
>>> print(ddd)
{'course': 182, 'age': 21}
>>> ddd['age'] = 23
>>> print(ddd)
{'course': 182, 'age': 23}
```



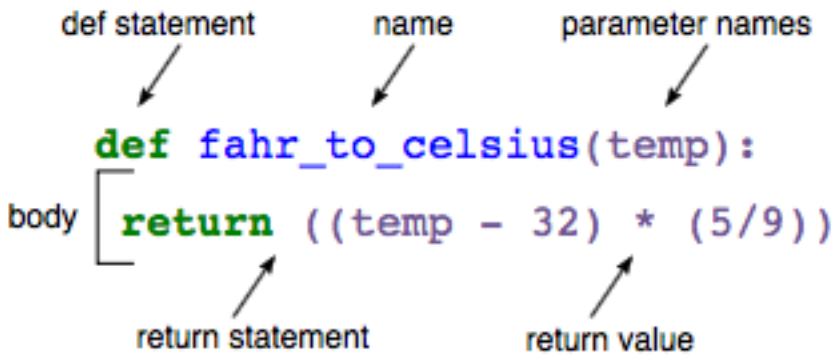
Sets

- Sets contains unordered collections of unique items. They are defined much like lists and tuples, except they use the curly brackets of dictionaries.

Functions

- A function is a named sequence of statements that performs a computation.
- When you define a function, you specify the name and the sequence of statements.
- Later, you can “call” the function by name.

- *08_def_1.py*
- *08_def_2.py*



https://swcarpentry.github.io/python-novice-inflammation/06-func/index.html

Control flow ***

- Control flow is where “the rubber really meets the road in programming.”
- With control flow, you can execute certain code blocks conditionally and/or repeatedly: these basic building blocks can be combined to create sophisticated programs.
  - conditional statements (“if”, “elif”, and “else”)
  - loop statements (“for” and “while” plus the accompanying “break”, “continue”, and “pass”)

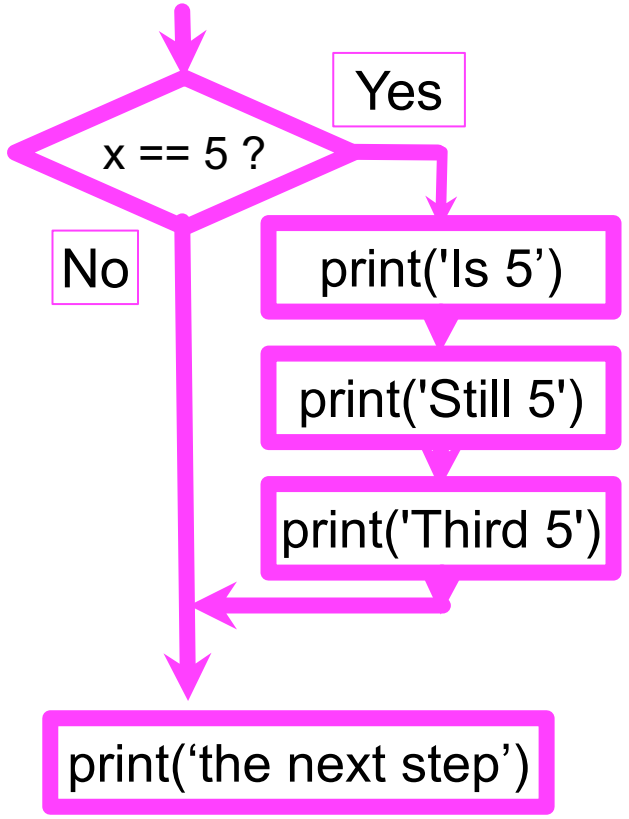
Comparison operators

- Boolean expressions ask a question and produce a Yes or No result which we use to control program flow
- Boolean expressions using comparison operators evaluate to True / False or Yes / No

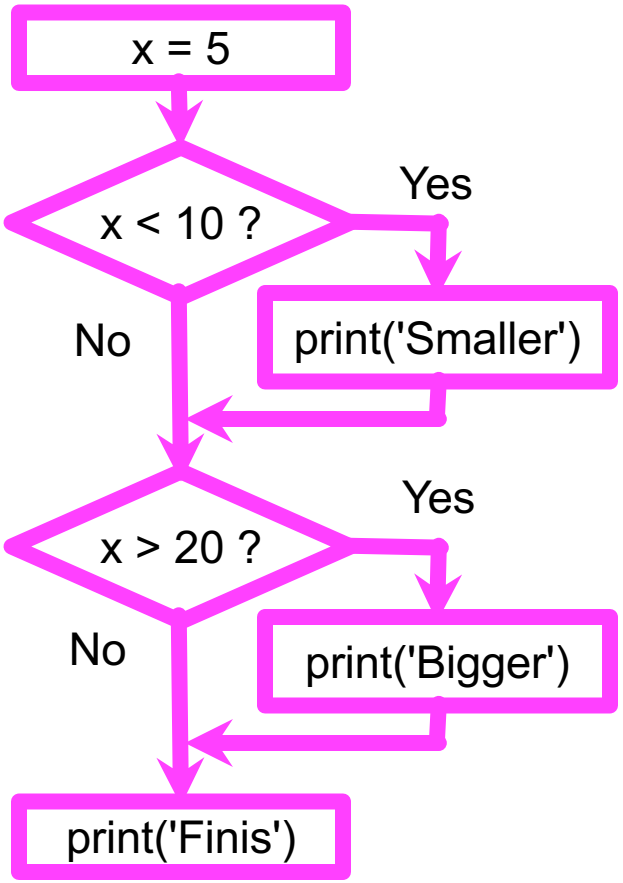
Python	Meaning
<	Less than
<=	Less than or Equal to
==	Equal to
>=	Greater than or Equal to
>	Greater than
!=	Not equal

One-way decisions

```
x = 5
print('Before 5')
if  x == 5 :
    print('Is 5')
    print('Is Still 5')
    print('Third 5')
print('the next step')
```



more one-way decisions



Program:

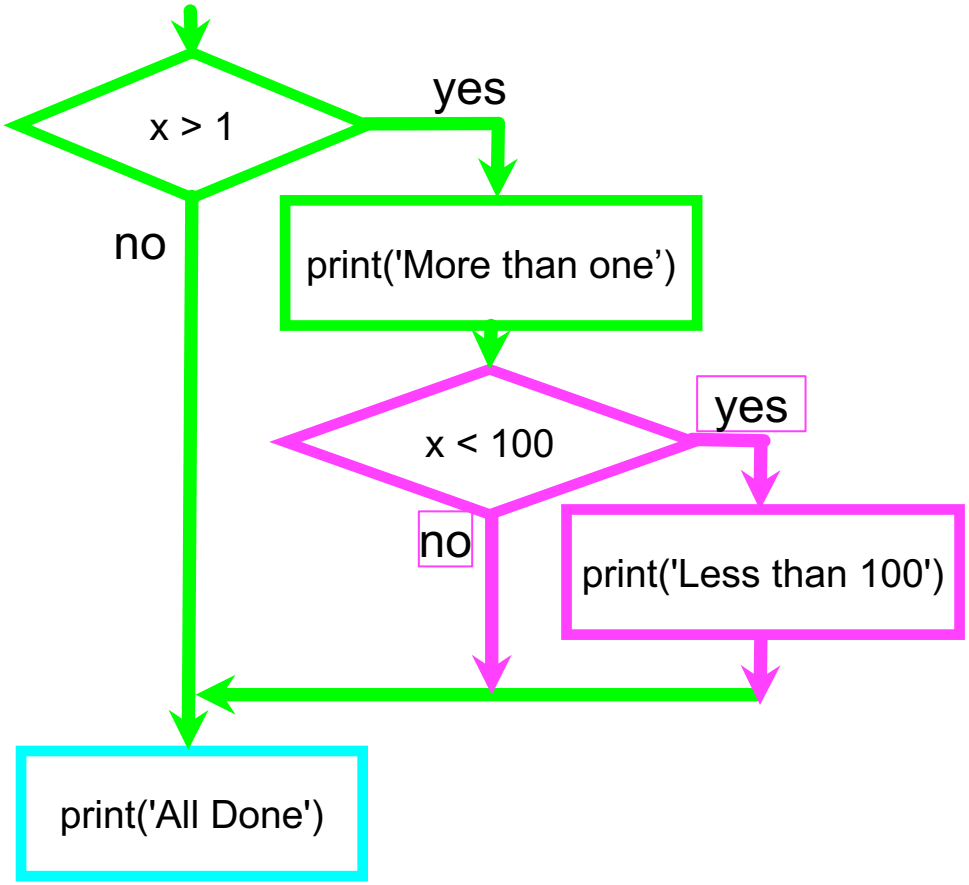
```
x = 5
if x < 10:
    print('Smaller')
if x > 20:
    print('Bigger')

print('Finis')
```

Output:

```
Smaller
Finis
```

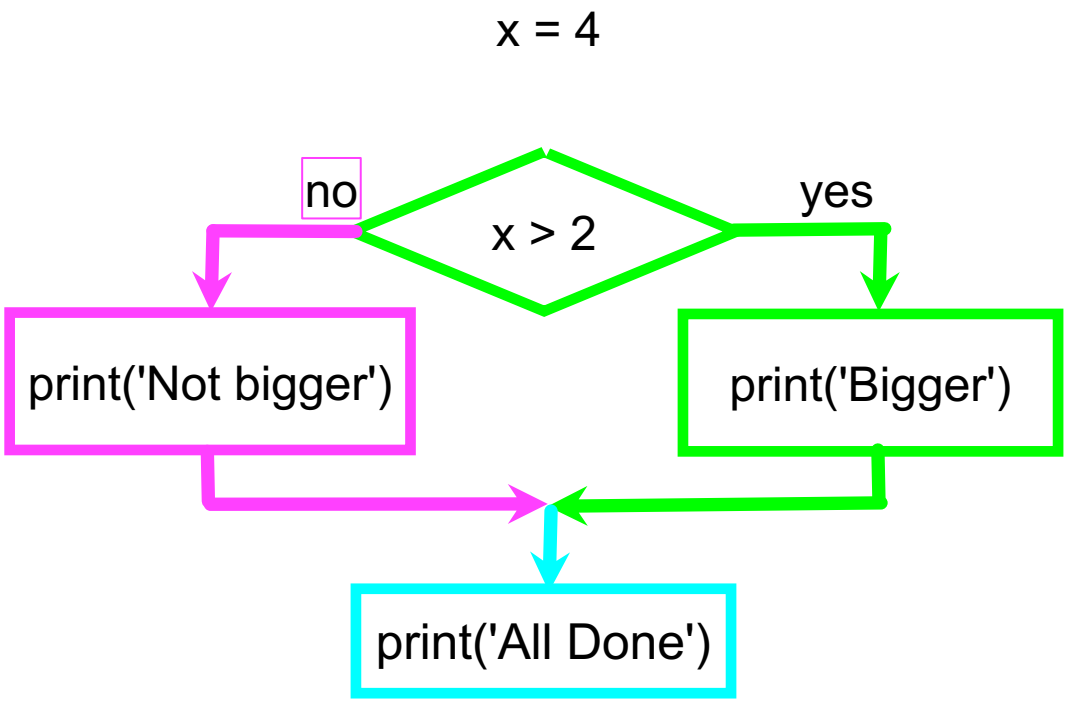
Nested decisions

```
x = 42
if x > 1 :
    print('More than one')
    if x < 100 :
        print('Less than 100')
print('All done')
```



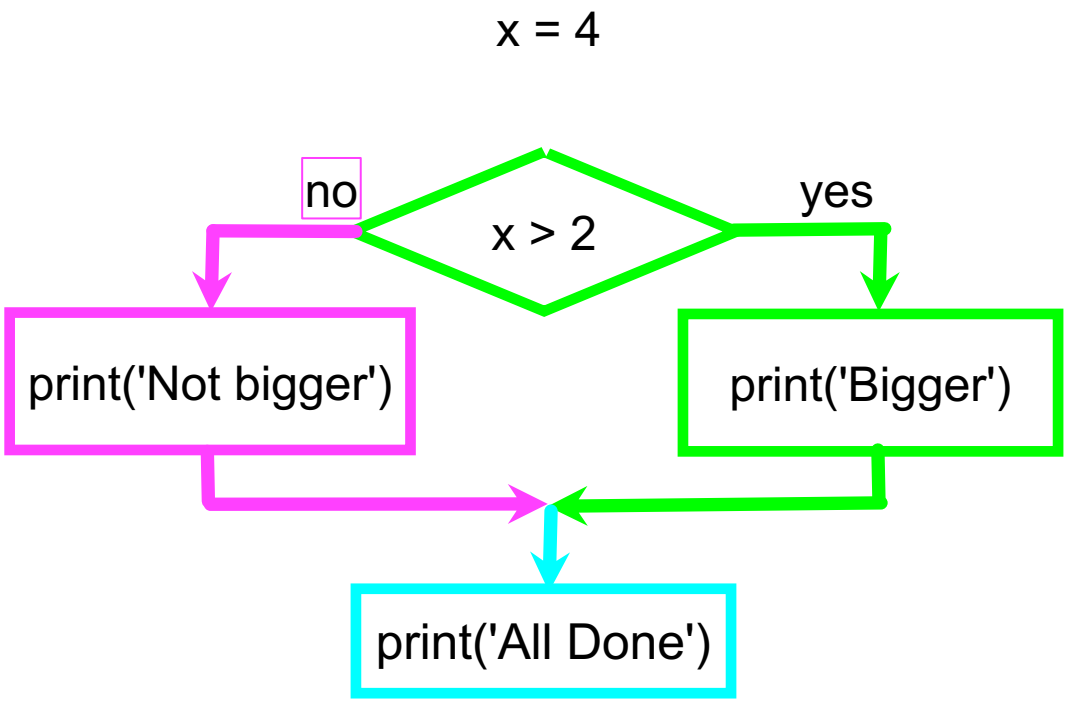
Two-way decisions (alternative execution)

- Sometimes we want to do one thing if a logical expression is true and something else if the expression is false
- It is like a fork in the road - we must choose one or the other path but not both.



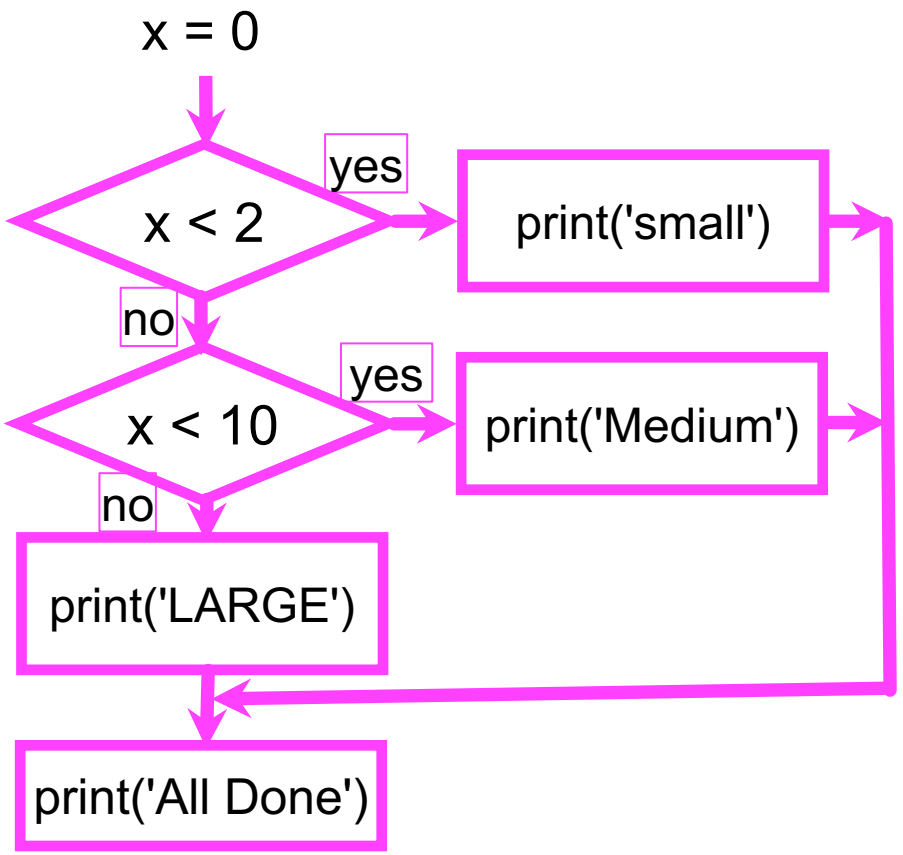
Two-way decisions: ***else***

```
x = 4

if x > 2 :
    print('Bigger')
else :
    print('Smaller')

print('All done')
```



Multi-way decisions: ***elif***

```
x = 0
if x < 2 :
    print('small')
elif x < 10 :
    print('Medium')
else :
    print('LARGE')
print('All done')
```



Multi-way decisions: *elif*

- There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn't have to be one.

The try/except structure

- In some tutorials (such as the Whirlwind tour Python), “try/except” is regarded a method for debugging, i.e., catching exceptions
- If the code in the try works - the except is skipped
- If the code in the try fails - it jumps to the except section

The try/except structure

```
try:
    print("this gets executed first")
except:
    print("this gets executed only if there is an error")
```

```
rawstr = input('Enter a number:')
try:
    ival = int(rawstr)
except:
    ival = -1

if ival > 0 :
    print('Nice work')
else:
    print('Not a number')
```

Loops & Iterations

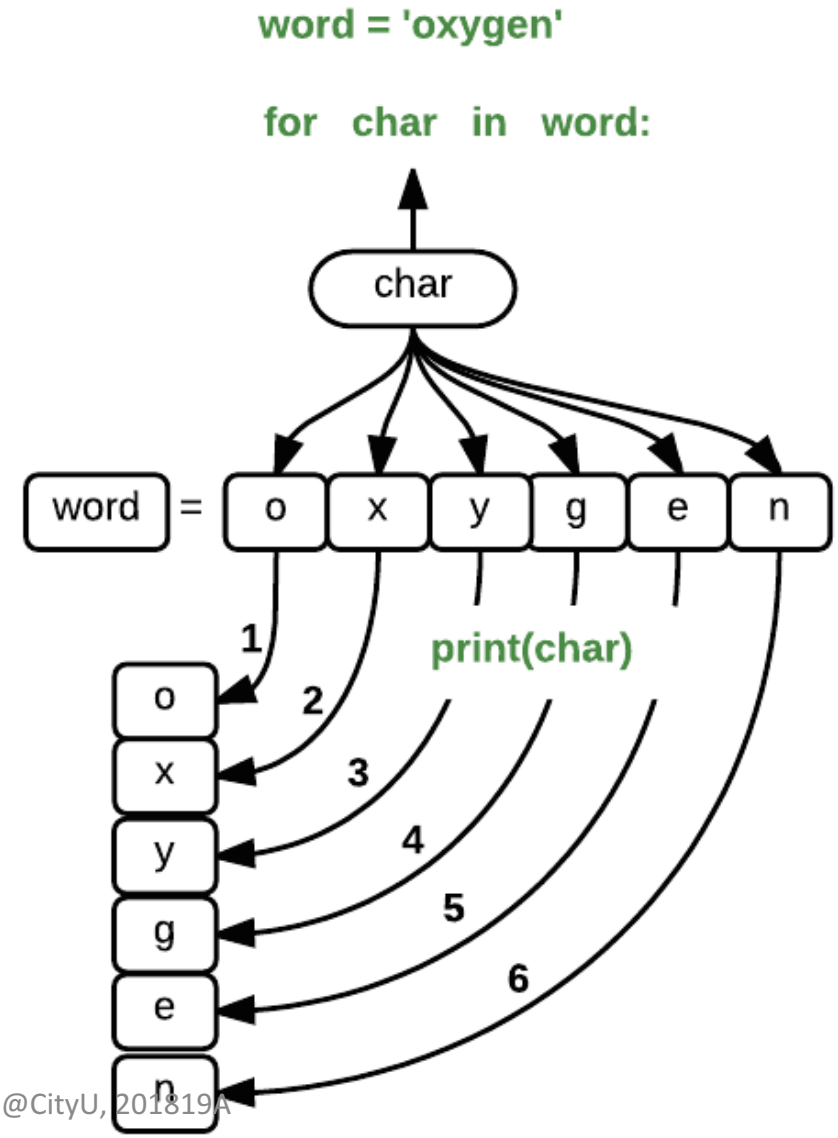
- Computers are good at doing repetitive work.
  - “for loops”
  - “while loops”
  - “break” and “continue”

For loops

- Sometimes we want to loop through a set of things such as a list of words, the lines in a file, or a list of numbers. When we have a list of things to loop through, we can construct a definite loop using a for statement.

For loops

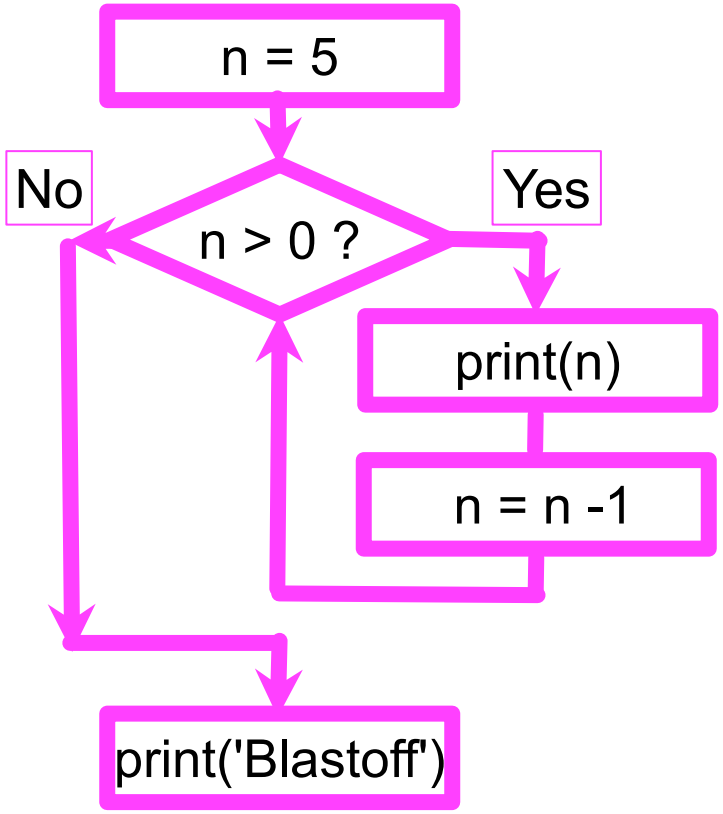
- **for** variable **in** collection: do things **with** variable



While loops

- A while statement is executed in the following sequence:
  - Evaluate the condition, yielding True or False.
  - If the condition is false, exit the while statement and continue execution at the next statement.
  - If the condition is true, execute the body and then go back to step 1.

While loops



Program:

```
n = 5
while n > 0 :
    print(n)
    n = n – 1
print('Blastoff!')
print(n)
```

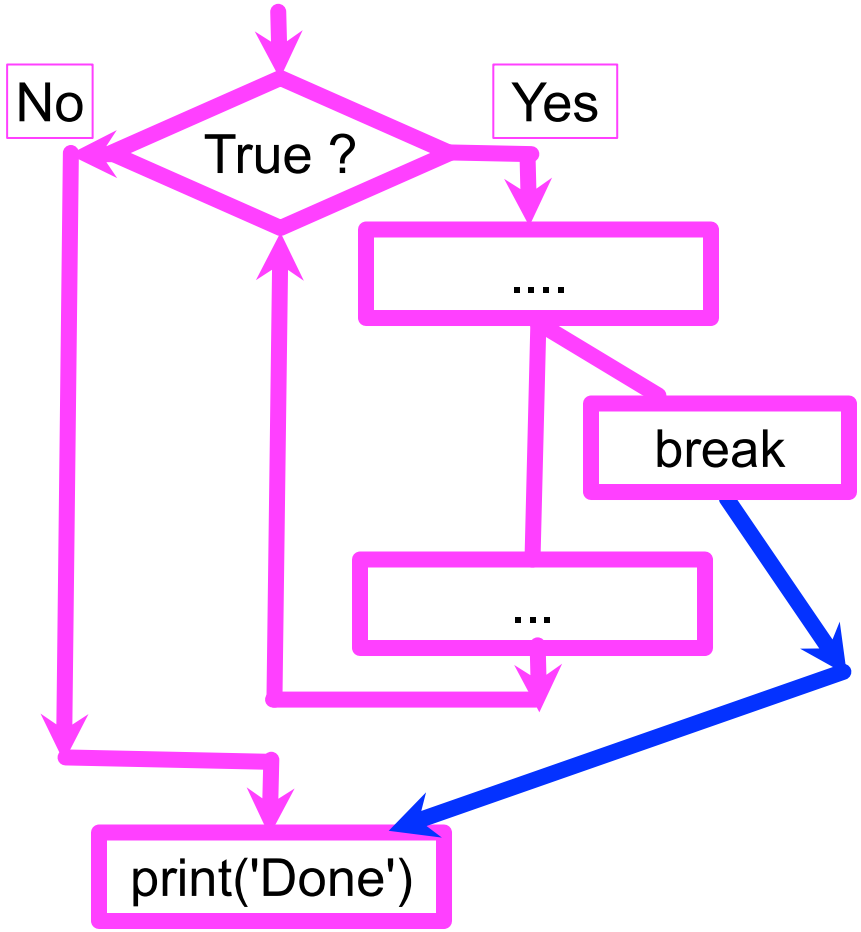
Output:

5
4
3
2
1
Blastoff!
0

Break

- The break statement ends the current loop and jumps to the statement immediately following the loop.

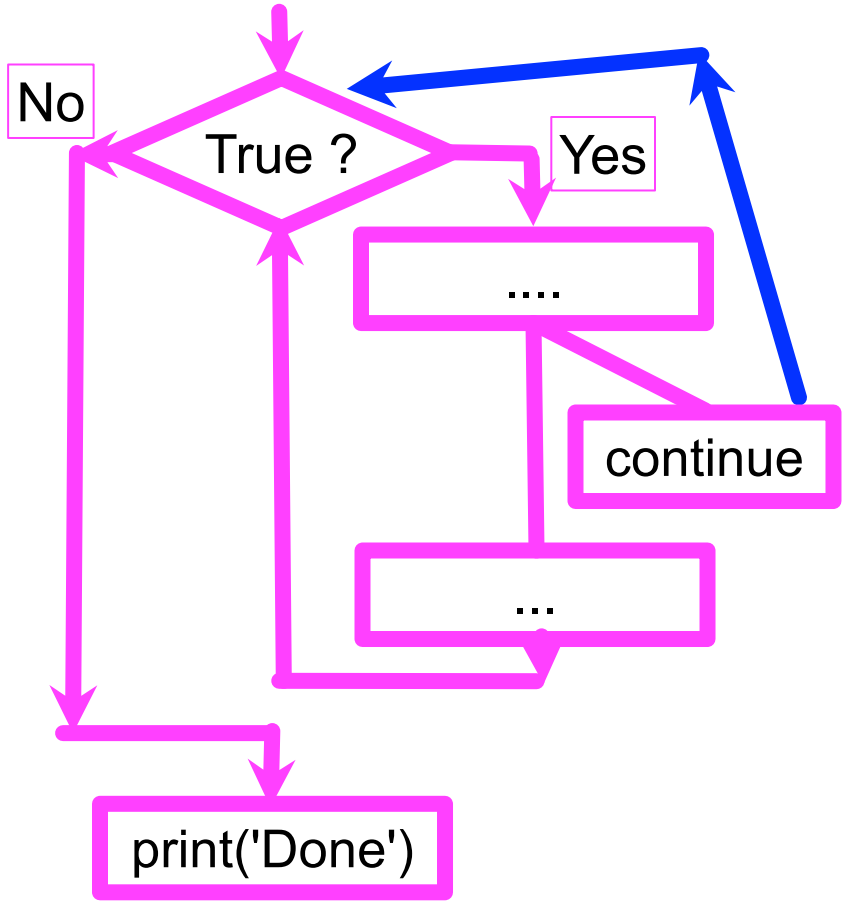
```
while True:
    line = input('> ')
    if line == 'done' :
        break
    print(line)
print('Done!')
```



Continue

- The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration.
- Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration.
- In that case you can use the continue statement to skip to the next iteration without finishing the body of the loop for the current iteration.

```
while True:
    line = raw_input('> ')
    if line[0] == '#' :
        continue
    if line == 'done' :
        break
    print(line)
print('Done!')
```



pass

- The “pass” statement does nothing.
- It can be used when a statement is required syntactically but the program requires no action.

Acknowledgements / Contributions

- Initial Development: Charles Severance, University of Michigan School of Information
- Compiled and revised for CityU HK COM5507 by Dr. Xinzhi Zhang

References

- https://swcarpentry.github.io/python-novice-inflammation/
- https://www.py4e.com/html3/05-iterations
- https://github.com/jakevdp/WhirlwindTourOfPython/blob/master/07-Control-Flow-Statements.ipynb