



WARWICK BUSINESS SCHOOL  
THE UNIVERSITY OF WARWICK

**For the  
Change  
Makers**

# Data Science with Python

**Day 1 (28<sup>th</sup> May)  
Information Systems and Management  
Warwick Business School  
Term 3, 2018-2019**



# Agenda

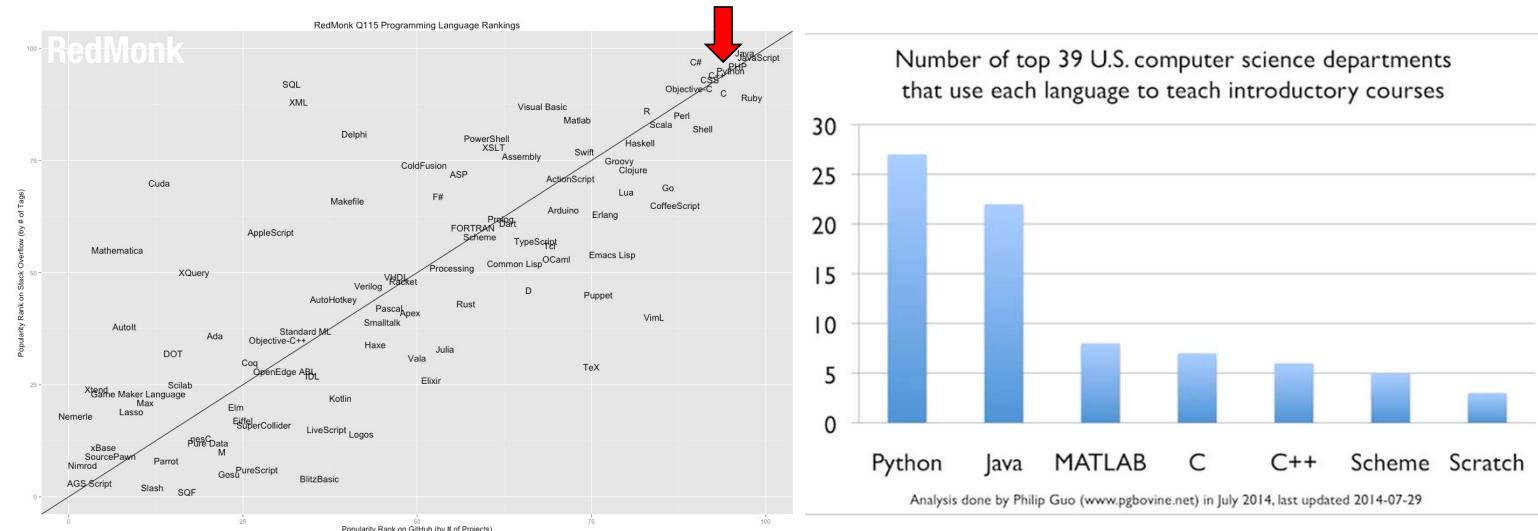
- Day 1:
  - Python basics
  - Data collection: web scraping using Requests and BeautifulSoup.
  - Data visualization: Seaborn
  - (Data cleaning and manipulation: Numpy and Pandas.)
- Day 2:
  - Data cleaning and manipulation: Numpy and Pandas.
  - Conventional machine learning: clustering using Scikit-Learn
  - Deep neural network: classification and time series forecasting using Keras

# Instructor

- Zhewei Zhang
  - ❑ Assistant professor at ISM
- Background: Ph.D. in Management Information Systems at Temple University, United States
- Related experience
  - ❑ Data Analytics
  - ❑ Computational research
  - ❑ System Analyst

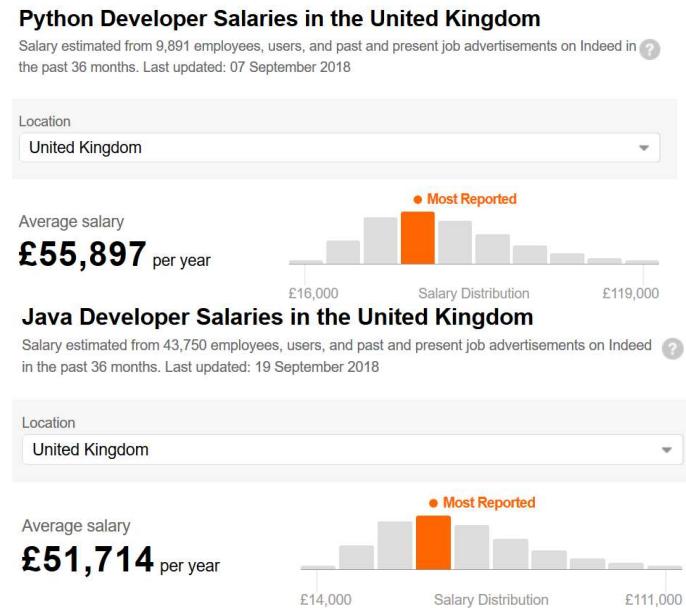
# Why learn Python?

- Python is one of the most “powerful” and widely used language, yet beginner friendly.



# Why take this module

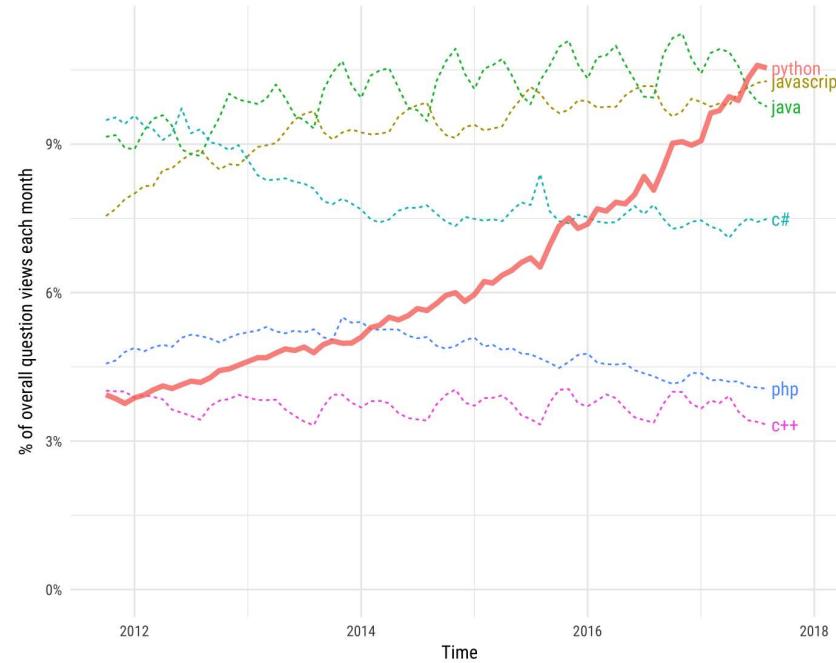
- Python is highly **needed** and paid well.
- 



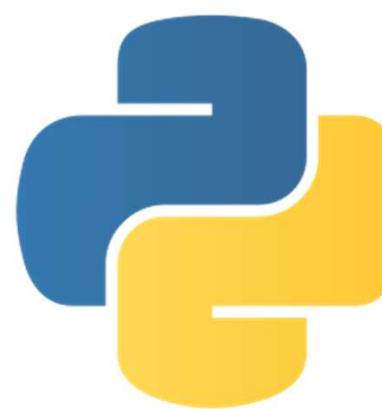
Source: <https://www.indeed.com/>

### Growth of major programming languages

Based on Stack Overflow question views in World Bank high-income countries



**After this workshop... You will**



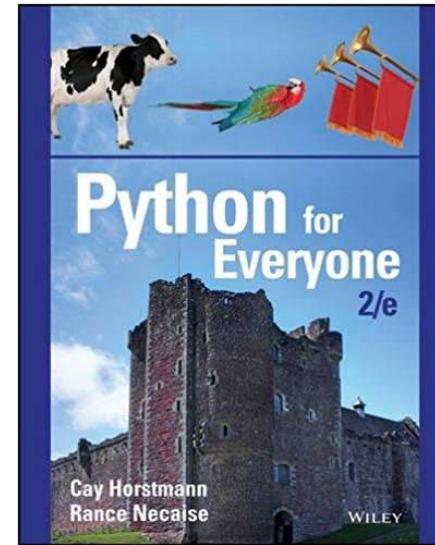
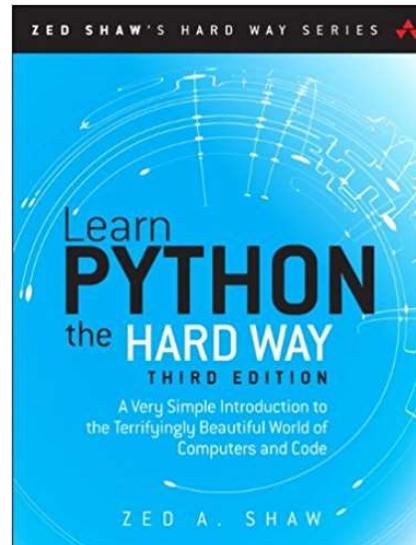
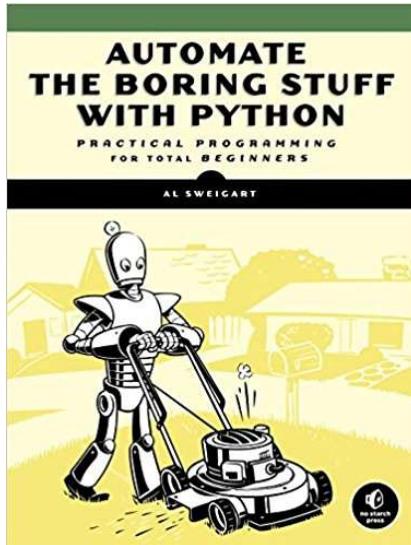
python  
powered

`print("Hello, world!")`

# Programming like Lego building



# Recommended textbooks



# Additional Resource

- How to Think Like a Computer Scientist: Learning with Python 3  
(<http://openbookproject.net/thinkcs/python/english3e/>)
- Python Official Documentation
- (<https://www.python.org/doc/>)
- Google, Stackoverflow, etc.

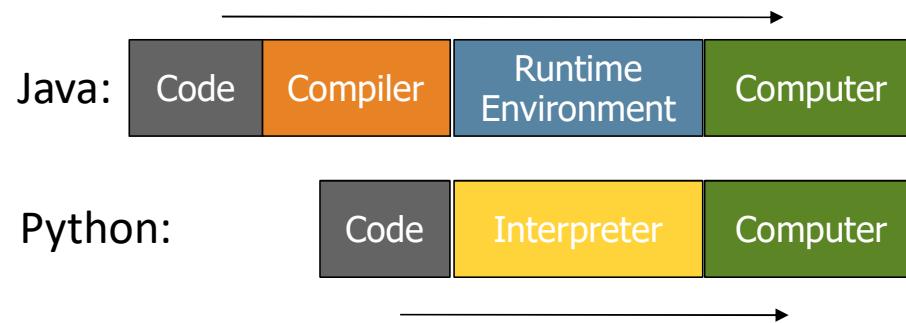
# Tools for Python coding

- Anaconda:
  - Spyder: IDE (integrated development environment)
  - Jupyter notebook: interactive
  - Python: 2 vs. 3
- Google Colab
- <https://repl.it>

# Python Basics

# Introduction to Python

- **Python is Interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
  - ❑ Not compiled like Java
  - ❑ Code is written and then directly executed by an interpreter
  - ❑ Type commands into interpreter and see immediate results



# Introduction to Python

- **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
  - ❑ Allows you to type commands one-at-a-time and see results
  - ❑ A great way to explore Python's syntax

```
Python 3.6.2 |Anaconda, Inc.| (default, Sep 19 2017, 08:03:39) [MSC v.
1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.1.0 -- An enhanced Interactive Python.

In [1]: print("Hello there!")
Hello there!

In [2]: print("How are you?")
How are you?

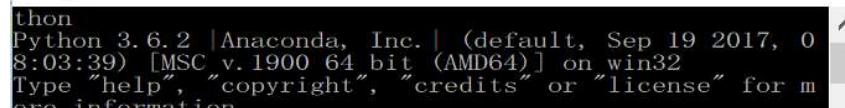
In [3]: |
```

# First Python program

- **Interactive mode programming**

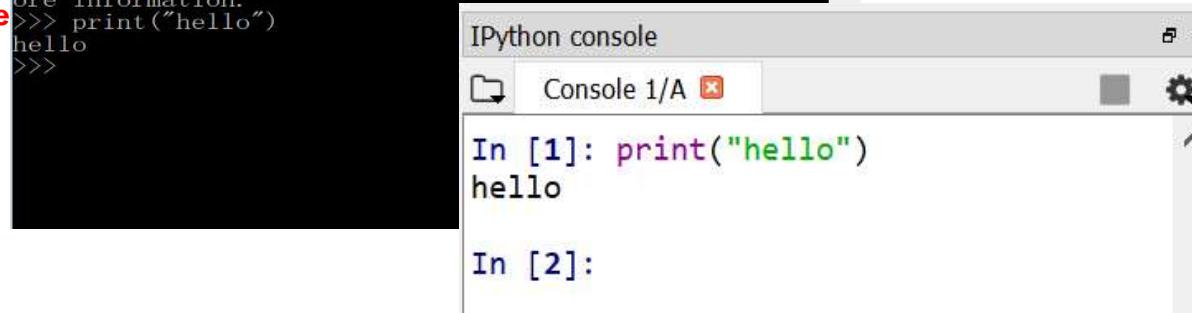
- When you enter into python IDE, you will see three greater signs. It means you are under the interactive mode.
  - Then you can write your code to let Python interpreter execute.

□ >>> print('Hello')



```
Python 3.6.2 |Anaconda, Inc.| (default, Sep 19 2017, 0  
8:03:39) [MSC v.1900 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license" for m  
ore information.  
>>> print("Hello")  
Hello
```

□ **This mode**



IPython console

Console 1/A

```
In [1]: print("Hello")
Hello

In [2]:
```

# First Python program

- Script mode programming (**mostly used**)
  - We will write a python program in a script (file). Python files have extension .py.
  - For example, test.py

```
print ("Hello, Python!")
x = 3
y = 5
sum = x+y
print ('x+y=',sum)
```

## The **print** statement

- Try the following statements on your own computer:

1. `print("hello world")`
2. `print(hello world)`
3. `print(hello world")`
4. `print("1+2")`
5. `print(1+2)`

## Types of program errors

- **Syntax** errors: errors due to the fact that the syntax of the language is not followed.
  - Easier to solve, code is wrong and error will be reported.
- **Semantic** errors: errors due to an improper use of program statements.
  - Harder to solve, code is valid and error may not be reported, code is not executed in the intended way.
- **Logical** errors: errors due to the fact that the specification is not respected.
  - Hardest to solve, code is correct and is executed in the intended way.

# Variables

- A **variable** is a named place in the memory where a programmer can store data and later retrieve the data using the variable “**name**”.
- Programmers need to choose the names of the variables.
  - You ‘define’ a variable by telling the program:
    1. What name you will use to refer to the variable
    2. The initial value of the variable
- You can change the contents of the variable later statement.



```
x = 12.2  
y = 14
```

12.2

y 14

Python uses equal sign "=" instead of "<-" to assign a value to a variable.

## Variables

- A **variable** is a named place in the memory where a programmer can store data and later retrieve the data using the variable “**name**”.
- Programmers need to choose the names of the variables.
  - You ‘define’ a variable by telling the program:
  - What name you will use to refer to it
  - The initial value of the variable
- You can change the contents of a variable in a later statement.

```
x = 12.2  
y = 14  
x = 100
```

x 12.2 100  
y 14

## Variable name rules

1. Must start with a letter or underscore \_
2. Only consist of letters, numbers, and underscores
3. Case sensitive
  - Different: smith Smith SMITH SmiTH
4. You can not use reserved words as variable names
  - smith \$smiths smith23 \_smith \_23\_
  - 23smith smith.23 a+b smiTHe -smith

No dot (.) in Python variable names.

## Variable name rules

1. Must start with a letter or underscore \_
2. Only consist of letters, numbers, and underscores
3. Case sensitive
  - Different: smith Smith SMITH SmiTH
4. You can not use reserved words as variable names
  - smith \$smiths smith23 \_smith \_23\_
  - 23smith smith.23 a+b smiTHe -smith

Green: Good var names   Red: Bad var names

## Special uses of underscore \_

- For storing the value of last expression in interpreter.

```
>>>1+2+3+4
```

```
>>>10
```

```
>>>_
```

```
>>>10
```

- To give special meanings

starting with `_` to indicate this is a “private” variable for internal use.

- Recommendation: not to start with “`_`” if you are unsure.

## Reserved Words

and	assert	break	class	continue
def	del	elif	else	except
exec	finally	for	from	global
if	import	in	is	lambda
not	or	pass	print	raise
return	try	while		

## Variables naming conventions

- Avoid using names that are too general or too wordy. Strike a good balance between the two.
  - Bad: data\_structure, my\_list, info\_map, dictionary\_for\_the\_purpose\_of\_storing\_data\_representing\_word\_definitions
  - Good: user\_profile, menu\_options, word\_definitions
- Avoid using “O”, “I”, or “L” as single character variable name.
- module\_name, package\_name, **ClassName**, method\_name, **ExceptionName**, function\_name, **GLOBAL\_CONSTANT\_NAME**, global\_var\_name, instance\_var\_name, function\_parameter\_name, local\_var\_name
- <https://www.python.org/dev/peps/pep-0008/#naming-conventions>
- [https://en.wikipedia.org/wiki/Naming\\_convention\\_\(programming\)](https://en.wikipedia.org/wiki/Naming_convention_(programming))

## Not Recommended Words

Data	Float	Int	Numeric	Oxphys
array	close	float	int	input
open	range	type	write	zeros
acos	asin	atan	cos	e
exp	fabs	floor	log	log10
pi	sin	sqrt	tan	

# First script

- Write your first Python script:
- Create two variables, one containing your first name and another containing your last name.
- Print your name with variables created.

## Value and data types

- The programmer (and the interpreter) can identify the type of data.
- You **do not need to explicitly** define or declare the type.
  - int x = 3 (for most other languages)
  - x = 3 (for Python)
  - Try type(x)
- Python has five standard data types
  - Numbers
  - String
  - List
  - Tuple
  - Dictionary

# Numbers

- Python supports four different numerical types:
  - ❑ int (signed integers, similar as Integer in R)
  - ❑ long (long integers, they can also be represented in octal and hexadecimal)
  - ❑ float (floating point real values, similar as Numeric in R)
  - ❑ complex (complex numbers, similar as Complex in R)

<b>int</b>	<b>float</b>	<b>complex</b>
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j
080	32.3+e18	.876j
-0490	-90.	-.6545+0j
-0x260	-32.54e100	3e+26j
0x69	70.2-E12	4.53e-7j

## Strings

- Strings in Python are identified as a contiguous set of characters represented in the single or double quotes.
- *Subsets of strings can be taken using the slice operator ([ ] and [ : ]) with indexes starting at 0 in the beginning of the string.*
- Try:
  - `str = 'Hello World!'`
  - `print (str)`
  - `print (str[0])`
  - `print (str[2:5])`                    `print (str[2:])`

## Get variable type

- If you are not sure what type a variable has, the interpreter can tell you by using **type()**.
- Try:
  - `type('Hello, world!')`
  - `type(17)`
  - `type(3.2)`
  - `type('4.7')`

Similar as `class()` in R.

## Type matters

- Python knows what “type” everything is.
- Some operations are prohibited.
  - You can not add a string to an integer.
  - Try:
    - . a = '123'
    - . b = a + 1

## Type conversions

- You can use `int()` and `float()` to convert between strings and integers/floating points.
- You will get an error if the string does not contain numeric characters.
  - Try:

```
a = "123"
b = a + 1
b = int(a)
print(b)
b = a + str(1)
float(a)
```

# Operators

- Operators are special symbols that represent computations like addition and multiplication.
- The values the operator uses are called **operands**.
- When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed.
- Types of common operator
  - Assignment operators
  - Arithmetic operators
  - Comparison (Relational) operators
  - Logical operators
  - Membership operators
  - Identity operators

# Arithmetic operators

- Assume variable a holds 10 and variable b holds 20.

Operator	Description	Example
+	Addition	Adds values on either side of the operator. $a + b = 30$
-	Subtraction	Subtracts right hand operand from left hand operand. $a - b = -10$
*	Multiplication	Multiplies values on either side of the operator $a * b = 200$
/	Division	Divides left hand operand by right hand operand $b / a = 2$
%	Modulus	Divides left hand operand by right hand operand and returns remainder $b \% a = 0$
**	Exponent	Performs exponential (power) calculation on operators $a**b = 10 \text{ to the power } 20$
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	$9//2 = 4$ and $9.0//2.0 = 4.0$

Floor division is not round up quotient

# Examples

- Handy to use for making change:

- pennies = 1729
- dollars = pennies / 100 # 17
- cents = pennies % 100 # 29

Table 3 Floor Division and Remainder

Expression (where n = 1729)	Value	Comment
$n \% 10$	9	For any positive integer n, $n \% 10$ is the last digit of n.
$n // 10$	172	This is n without the last digit.
$n \% 100$	29	The last two digits of n.
$n \% 2$	1	$n \% 2$ is 0 if n is even, 1 if n is odd (provided n is not negative)
$-n // 10$	-173	-173 is the largest integer $\leq -172.9$ . We will not use floor division for negative numbers in this book.

# Operator precedence

- When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence.
- PEMDAS, which stands for Parentheses Exponents Multiplication Division Addition Subtraction.
  - ❑ Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want.
  - ❑ Exponentiation has the next highest precedence.
  - ❑ Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence
  - ❑ Operators with the same precedence are evaluated from left to right.

## Arithmetic operation

- Try following arithmetic operation statements:
  1.  $1 + 2 * 3 - 2 / 10$
  2.  $(1+2) * (3-2) / 10$
  3.  $1 + 2^{**}3 - 2 / 10$
  4.  $1 + 2 * 3 - 2 // 10$
  5.  $1 + 2 * 3 - 2 \% 10$
  6.  $1 + 2^{**}(3-2)//10$
- When you are unsure, always use parentheses to force the order, but double check the matching when multiple parentheses are used.

# **input() function**

- `input([prompt])` function will take input that the user type in the interactive window and store as value. An optional prompt can also be specified as function argument like `input("what's your name")`. The program will first output the prompt and wait for user's input.
- Try:
- `name = input("what is your name?")`
- `print(name)`
- You can read more here  
[\(https://docs.python.org/3/library/functions.html#input\)](https://docs.python.org/3/library/functions.html#input)

## Exercise

- Write a program using `input()` function and arithmetic operators to do the following tasks.
- 1) Obtain two integer inputs from users: X and Y
  - 2) Print out results for the equations  $(X+Y)/(X-Y)$  and  $(X-Y)^3$
  - 3) Print out the last digit of  $X+Y$

## Exercise 2

- Write a short Python script to 1) set the prices for different fruits (2.99 per box for apple, 2.49 per box for orange, and 1.99 per box for grape), and 2) calculate the total price based on the user's input.
- Sample output:

Please enter the number of boxes of apples in your cart: 2

Please enter the number of boxes of oranges in your cart: 3

Please enter the number of boxes of grapes in your cart: 3

Total price: 19.42

## Exercise 3

- Write a short Python script to calculate the cost to paint a circle shaped wall for two coats based on given information and user's input.
- $\pi=3.14$
- coverage of  $10m^2$  per litre of paint
- cost of paint per litre: £15 per litre
- Sample output:

Please enter the circle's radius in metre: 3

The cost to paint is 84.78

## Quick Review

- print()
- comment
- Variables
- Naming
- Assignment
- Data type and conversion
- Arithmetic operation

# Conditional Statements

## Program steps or flow

- Like a recipe or installation instructions, a program is a **sequence** of steps to be done in order.
- **Some steps are conditional - they may be skipped.**
- Sometimes a step or group of steps are to be **repeated**.
- Sometimes we store a set of steps to be used over and over as needed several places throughout the program. We call it **function**.

## Condition Statement

- A computer program often needs to make decisions based on input, or circumstances
- For example, it is illegal for people under 18 to buy alcohol in UK.  
For cashier,
  - If the person looks younger than 25, a ID check may be performed by the cashier.

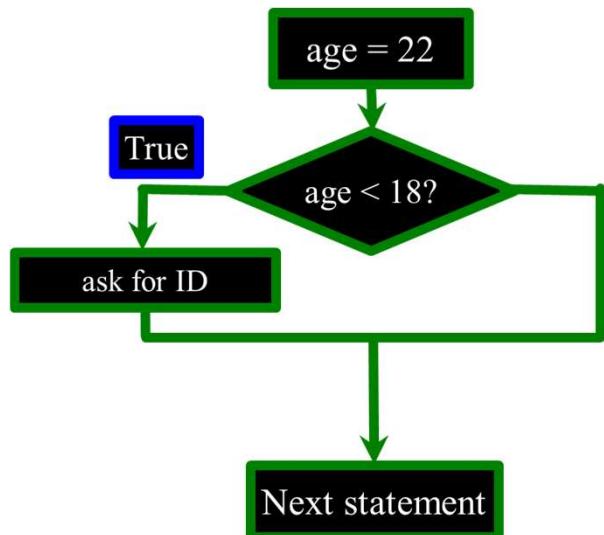
## The **if** Statement

- The if statement allows a program to carry out different actions depending on the nature of the data to be processed (**condition**).
- The keyword of the if statement is:

**if**

## Flowchart of **if** statement

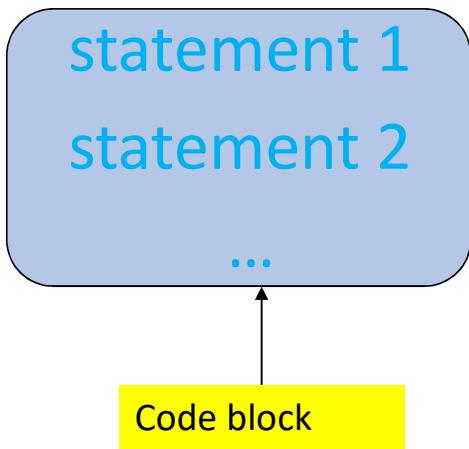
- The statement will only be executed if the condition is **true**.



```
age = 22  
if age < 18:  
    id_check()  
    checkout()
```

# if statement syntax

if condition:



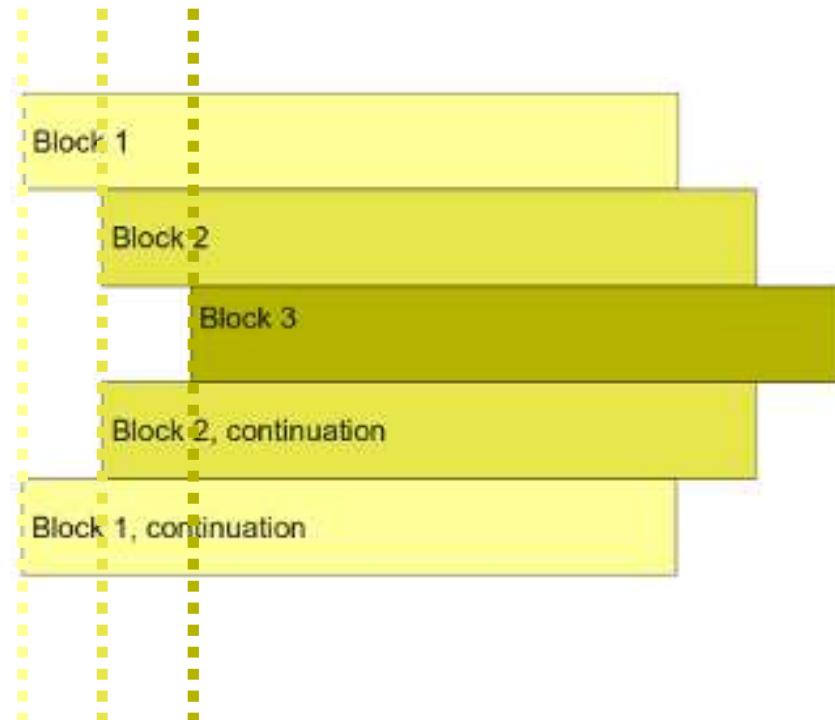
- Condition is a *expression* that returns a value **True** or **False**.
- A group of statements will be executed in sequence while condition is true.

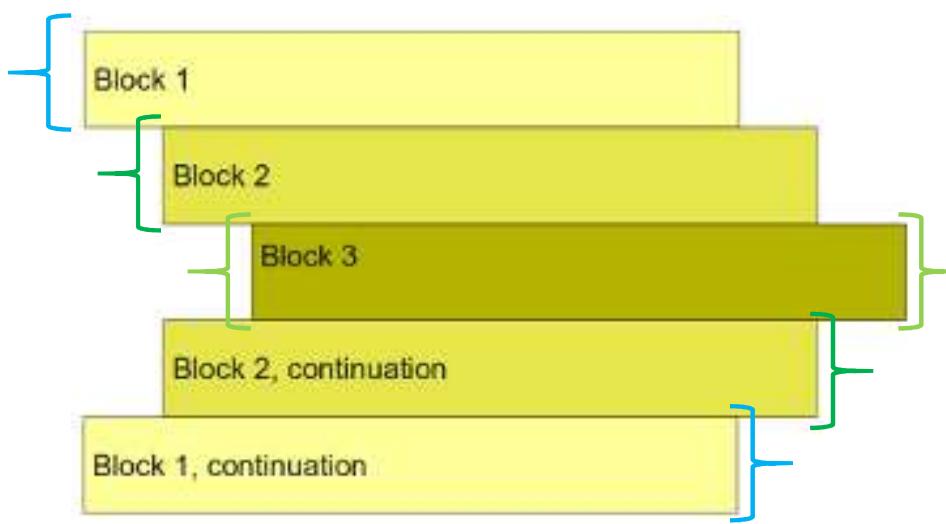
# Structuring with Indentation

- A **block** is a group of statements.
  - ✓ at least one statement and of declarations for the block
  - ✓ blocks can contain blocks as well, so we get a nested block structure
  - ✓ group of statements to be treated as if they were one statement
- Python programs get structured through **indentation**
  - ✓ code blocks are declared by their indentation
  - ✓ All statements with the same distance to the left belong to the same block of code
  - ✓ If a block has to be more deeply nested, it is simply indented further to the right.

A big difference between Python and R is how statements are structured. Python to use indentation while R use {}

## Nested blocks

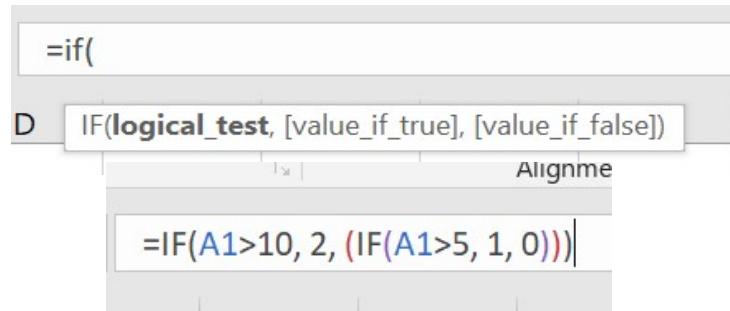




## Nested blocks

- Block 2

Statement 1
Statement 2 (group of statements in block3)
Statement 3



```
if A1>10:  
    print(2)  
else:  
    if A1 > 5:  
        print(1)  
    else:  
        print(0)
```

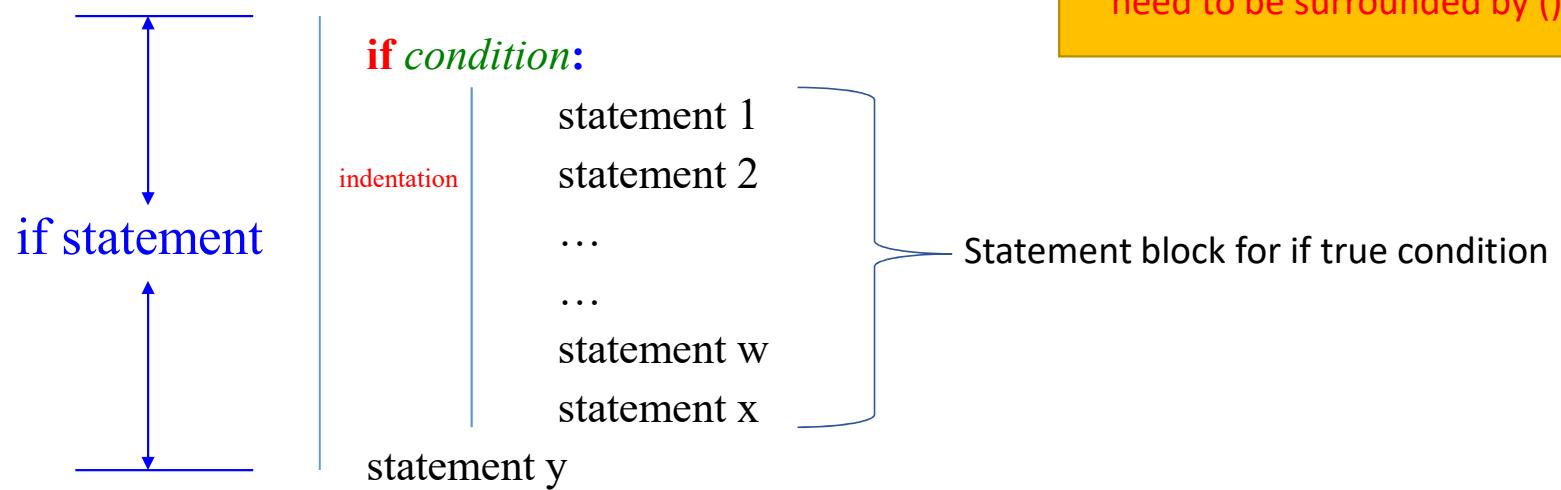
## Think about begin/end blocks

```
age = int(input("age?"))
if age < 18:
    print("underage!")
    print("show your ID!")
print("Finished checkout")
```

- Increase / maintain after if
- Decrease to indicate end of block

```
→ age = int(input("age?"))
→ if age < 18:
→     print("underage!")
→     print("show your ID!")
← print("Finished checkout")
```

## if statement syntax



## Exercise

- Try the following three scripts with age = 15 and age = 22.

```
age = int(input("age?"))
if age < 18:
    print("underage!")
    print("show your ID!")
print("Finished checkout")
```

```
age = int(input("age?"))
if age < 18:
    print("underage!")
    print("show your ID!")
print("Finished checkout")
```

```
age = int(input("age?"))
if age < 18:
    print("underage!")
        print("show your ID!")
print("Finished checkout")
```

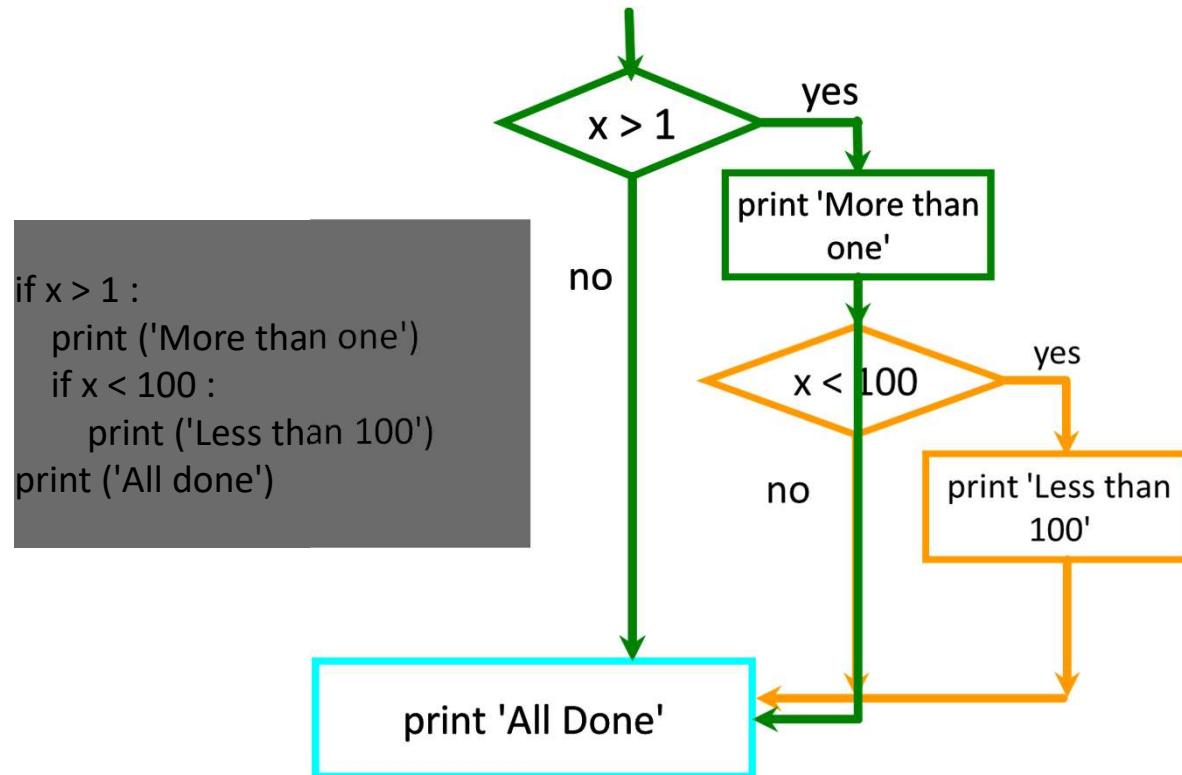
## Compound Statements

- **Compound statements** span multiple lines and consist of a *header* and a statement block
  - The if statement is an example of a compound statement
- Compound statements require a colon “:” at the end of the header.
- The statement block *starts on the line after the header* and *ends at the first statement with the same indentation as header*.
  - *In spyder, use Tab to indent and Shift+Tab to unindent*

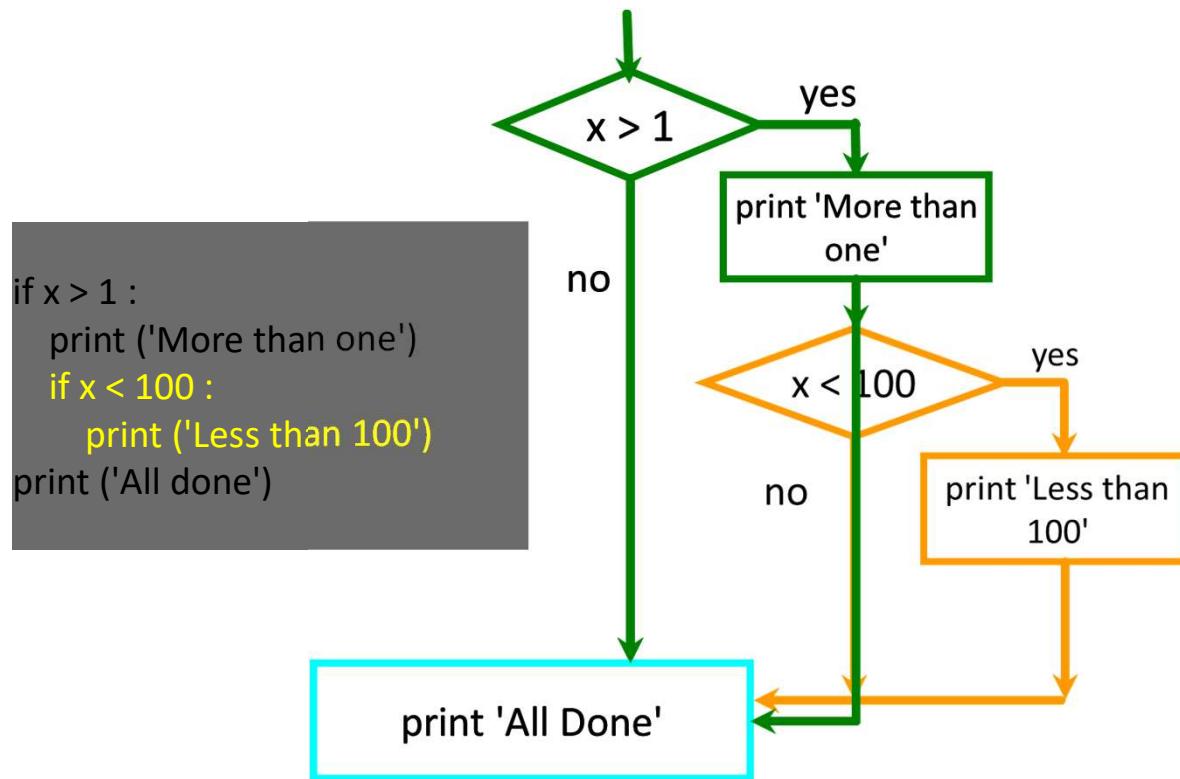
## Nested conditions

- You may want to check another condition when one condition is already true.
  - If your customer looks younger than 25, if she/he can show an ID
- The branch of if statement can also be another condition statement.

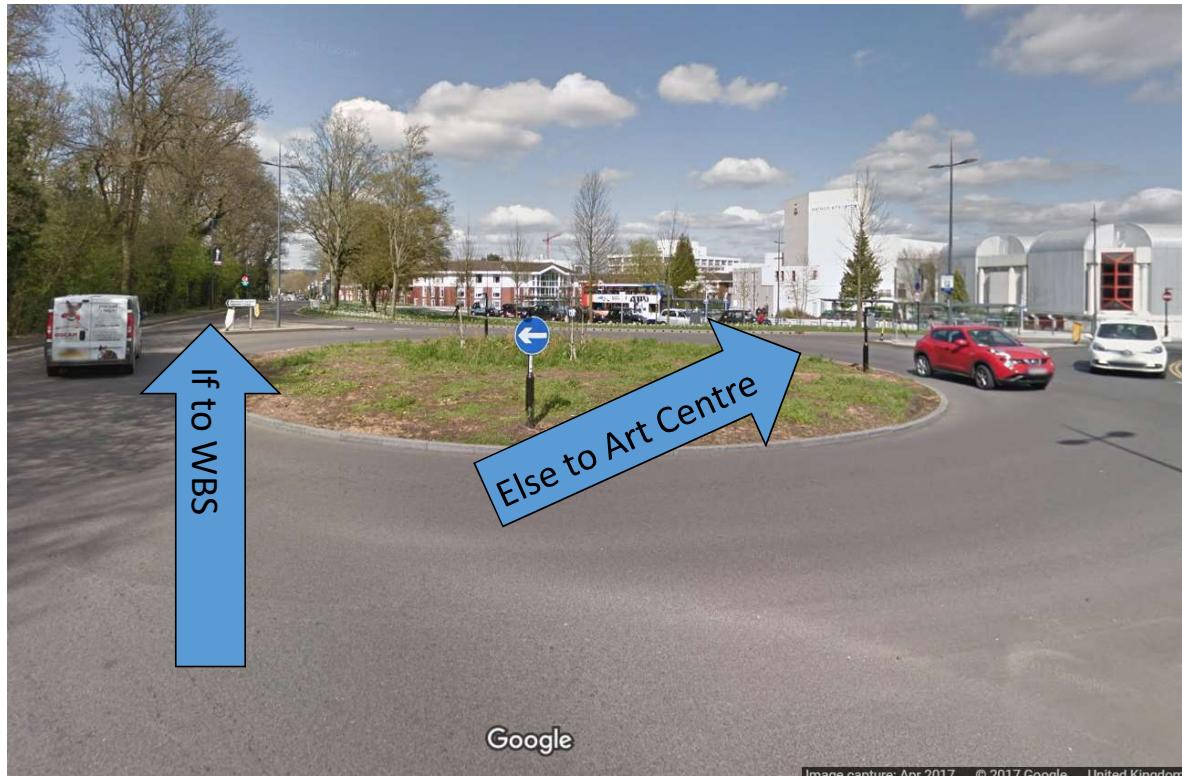
## Nested if-statements



## Nested if-statements



# If-else

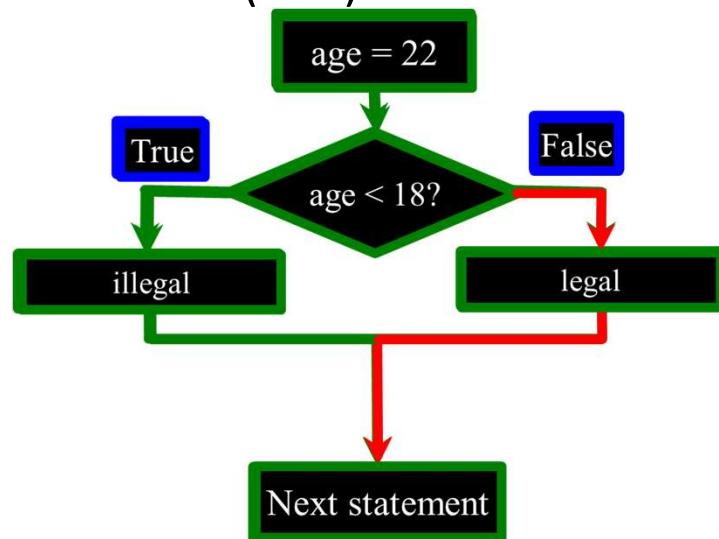


## if-else statement

- 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

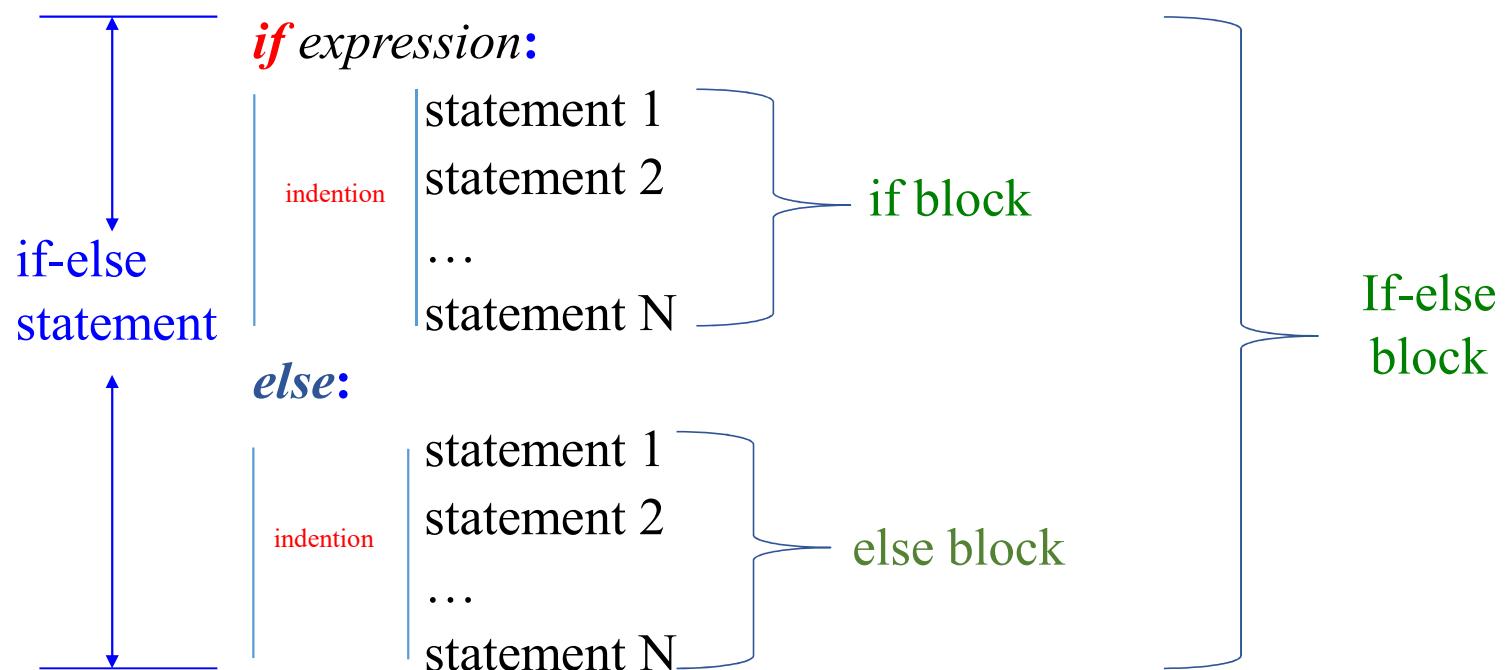
## Flowchart of if...else

- One of the two branches is executed once
  1. True branch (if)
  2. False branch (else)



```
age = 22  
if age < 18:  
    print ("illegal")  
  
else:  
    print("legal")
```

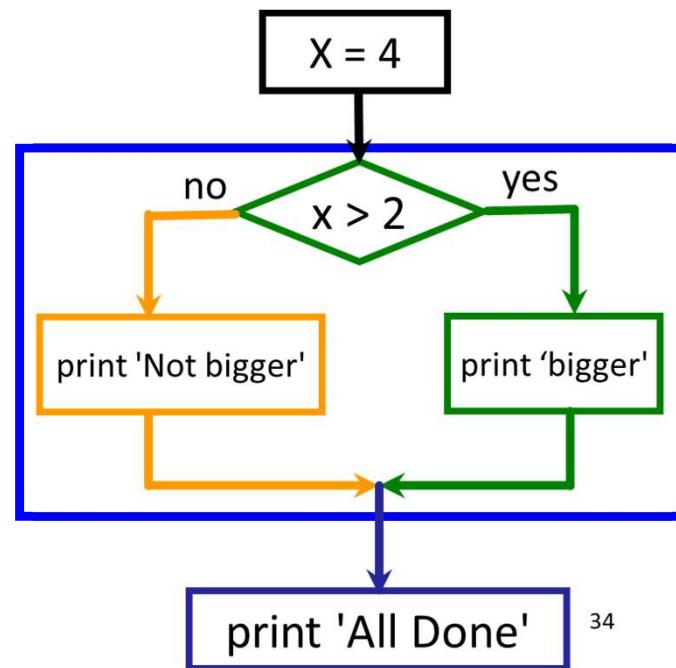
## if-else statement syntax



## if-else statement

```
x = 4
```

```
if x > 2 :  
    print ('Bigger')  
else :  
    print ('Not bigger')  
  
print ('All done')
```



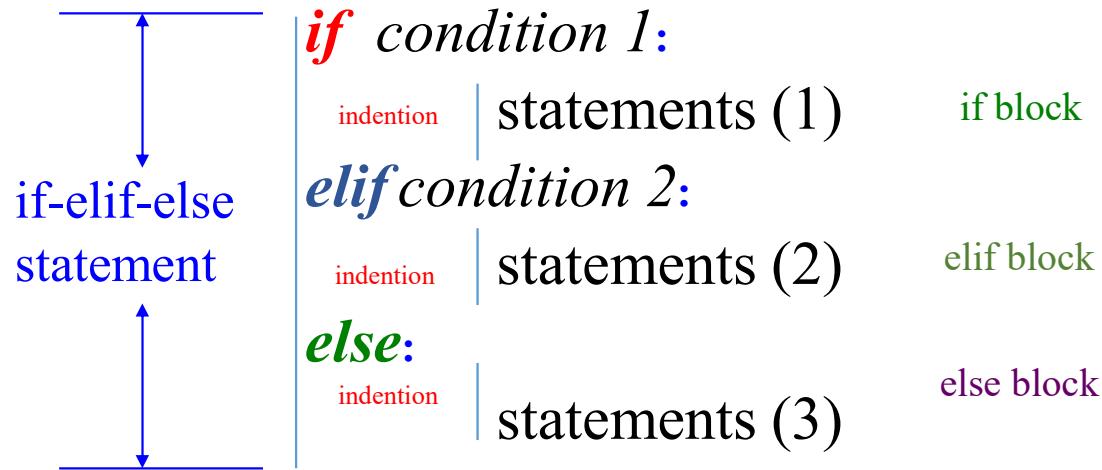
# Multi-conditions



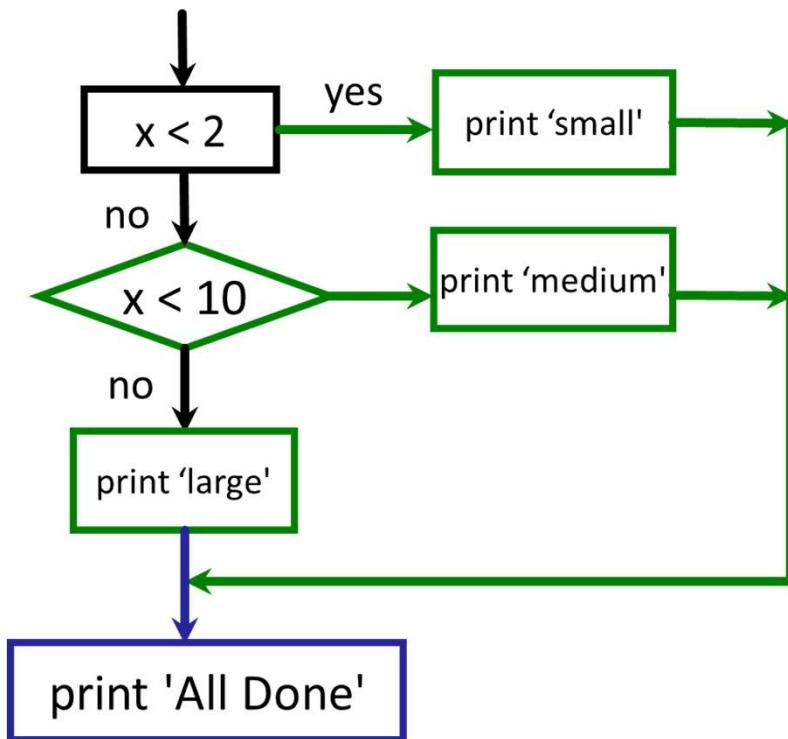
## if-elif-else statement

- Sometimes there may be more than two options.
- In R or many other languages, you can use '**switch**' statement. In Python, we don't have that. Instead, we then can use **if-elif** (else if)-**else** statement.

## if-elif-else statement syntax



## Multi-conditions



```
x = 5
if x < 2 :
    print ('Small')
elif x < 10 :
    print ('Medium')
else :
    print ('Large')
print ('All done')
```

## Variations (1): if...elif (no else)

- Syntax:

*if* expression 1:  
    statements (1)  
*elif* expression 2:  
    statements (2)  
statements...

```
# No Else
x = 5
if x < 2 :
    print ('Small')
elif x < 10 :
    print ('Medium')
print ('Large')
print ('All done')
```

## Variations (2): if...elif...elif.....else

- Syntax:

***if*** expression 1:  
statements (1)

***elif*** expression 2:  
statements (2)

***elif*** expression 3:  
statements (3)

...

***else***:  
statements (N)

```
if x < 2 :  
    print ('Small')  
elif x < 10 :  
    print ('Medium')  
elif x < 20 :  
    print ('Big')  
elif x < 40 :  
    print ('Large')  
elif x < 100:  
    print ('Huge')  
else :  
    print ('Ginormous')
```

What's the difference between  
'switch' statement and elif  
statement?

# Exercise

- Get a user's input of an integer.
- Check if the number can be divided by 4. If yes, print something; if no, print the remainder.

## Nested vs. Multi-Conditions

- Nested conditions: child condition will only be tested if parent condition is true.
- Multi-conditions: later condition will only be tested if previous condition is not true.
- They may be converted to each other.

## Examples

```
if x < 100 :  
    if x < 20 :  
        if x < 10 :  
            if x<2:  
                print ('Small')  
            else:  
                print ('Medium')  
        else:  
            print ('Big')  
    else :  
        print('Huge')  
else:  
    print ('Ginormous')
```

```
if x < 2 :  
    print ('Small')  
elif x < 10 :  
    print ('Medium')  
elif x < 20 :  
    print ('Big')  
elif x < 100:  
    print ('Huge')  
else :  
    print ('Ginormous')
```

## Nested cases

```
if x < 2 :  
    print ('Small')  
elif x < 10 :  
    print ('Medium')  
elif x < 20 :  
    print ('Big')  
elif x < 100:  
    print ('Huge')  
else :  
    print ('Ginormous')
```

```
if x < 2 :  
    print ('Small')  
else:  
    if x < 10 :  
        print ('Medium')  
    elif x < 20 :  
        print ('Big')  
    elif x < 100:  
        print ('Huge')  
    else :  
        print ('Ginormous')
```

# Puzzles

- Which statement will never print?

- 

```
if x < 2 :  
    print ('Below 2')  
elif x >= 2 :  
    print ('Two or more')  
else :  
    print ('Something else')
```

```
if x < 2 :  
    print ('Below 2')  
elif x < 20 :  
    print ('Below 20')  
elif x < 10 :  
    print ('Below 10')  
else :  
    print ('Something else')
```

## Statement in multi-lines

- Sometimes, the statement may be too long to display in one line. We can write the statement across different lines.
1. Statements may span multiple lines if you're continuing an open syntactic pair.
    - Continue typing a statement on the next line if you're coding something enclosed in a (), {}, or [] pair
    - Continuation lines can start at any indentation level, try to make them align vertically for readability if possible.
  2. Statements may span multiple lines if they end in a backslash

# Examples

```
x = 1  
y = 2  
z = 3  
print(x, y, z)  
print(x,  
      y,  
      z)  
print(x,  
      y,  
      z)
```

```
x = 1  
y = 2  
z = 3  
sum1 = x + y + z  
print(sum1)  
sum2 = (x + y  
        + z)  
print(sum2)  
  
sum3 = x + y \  
       + z  
print(sum3)
```

# Exercise

- Write a program using conditions and user input to generate the outcome of the rock, paper, scissor game. Example output should look like
- Player 1 : rock
- Player 2 : paper
- Player 2 wins

## Boolean expressions

- A Boolean expression is an expression that returns a value of either **true** or **false**.

$x == y$	x is equal to y
$x != y$	x is not equal to y
$x > y$	x is greater than y
$x < y$	x is less than y
$x >= y$	x is greater than or equal to y
$x <= y$	x is less than or equal to y

```
x = 5  
y = 7  
x >= y  
x == y  
x != y  
z = (x<7)  
print(z)
```

- Try following scripts and compare the results:

```
x = 4
if x > 2 :
    print ('Bigger')
else :
    print ('Smaller')

print ('All done')
```

```
x = 4
if True :
    print ('Bigger')
else :
    print ('Smaller')

print ('All done')
```

```
x = 4
if False:
    print ('Bigger')
else :
    print ('Smaller')

print ('All done')
```

## Logical operators

- Logical Operators are used to combine two or more conditions and perform the logical operations.
- There are three logical operators: *and*, *or*, and *not*.
  - ❑ *and* expression is true if and only if both operands are true
  - ❑ *or* expression is true if either one of operands is true
  - ❑ *not*: reverse the value

## Exercise

- Try following statements in sequence:
- $x = 5$
- $y = 7$
- $x+y>10$
- $x-y<0$
- $x+y>10$  and  $x-y<0$
- $x+y>10$  or  $x-y<0$
- not  $(x+y>10)$

## Exercise

```
x = 5  
y = 7  
if x+y>10 and x-y<0:  
    print('nice work')
```

```
x = 5  
y = 7  
if x+y>10:  
    if x-y<0:  
        print('nice work')
```

## Exercise

```
x = 5  
y = 7  
if x+y>10 or x-y<0:  
    print('nice work')
```

```
x = 5  
y = 7  
if x+y>10:  
    print('nice work')  
elif x-y<0 :  
    print('nice work')
```

# Truth Table

Assume x and y are Boolean expression

<b>x</b>	<b>and</b>	<b>y</b>	<b>Returns</b>
True	and	True	True
True	and	False	False
False	and	True	False
False	and	False	False

Assume a=3, b=10

<b>x</b>	<b>and</b>	<b>y</b>	<b>Returns</b>
a > 1	and	b > 2	True
a > 1	and	b < 2	False
a > 4	and	b > 2	False
a > 4	and	b < 2	False

<b>x</b>	<b>or</b>	<b>y</b>	<b>Returns</b>
True	or	True	True
True	or	False	True
False	or	True	True
False	or	False	False

<b>x</b>	<b>or</b>	<b>y</b>	<b>Returns</b>
a > 1	or	b > 2	True
a > 1	or	b < 2	True
a > 4	or	b > 2	True
a > 4	or	b < 2	False

# Exercises

- 1. True and True
- 2. False and True
- 3. `1 == 1` and `2 == 1`
- 4. `"test" == "test"`
- 5. `1 == 1` or `2 != 1`
- 6. True and `1 == 1`
- 7. False and `0 != 0`
- 8. True or `1 == 1`
- 9. `"test" == "testing"`
- 10. `1 != 0` and `2 == 1`

## Exercises

- 11. "test" != "testing"
- 12. "test" == 1
- 13. not (True and False)
- 14. not (1 == 1 and 0 != 1)
- 15. not (10 == 1 or 1000 == 1000)
- 16. not (1 != 10 or 3 == 4)
- 17. not ("testing" == "testing" and "Zed" == "Cool Guy")
- 18. 1 == 1 and not ("testing" == 1 or 1 == 0)
- 19. "chunky" == "bacon" and not (3 == 4 or 3 == 3)
- 20. 3 == 3 and not ("testing" == "testing" or "Python" == "Fun")

## Exercises

- Write a program using condition statements and user input function to check if a given year is a leap year. If it is, the program should output “It is a leap year.”; if no, ~~the program should output the number of years to next leap year.~~ An example program looks as follows: (texts in blue are inputs from the user).
- Please enter the year: 1898
- This is not a leap year.
- ~~6 years to next leap year.~~

# Operator precedence

( ), {}, []
**
*, /, %, //
+, -
=

# Operator precedence

( ), {}, []
**
*, /, %, //
+, -
>, <, <=, >=
==, !=,
not
and
or
=

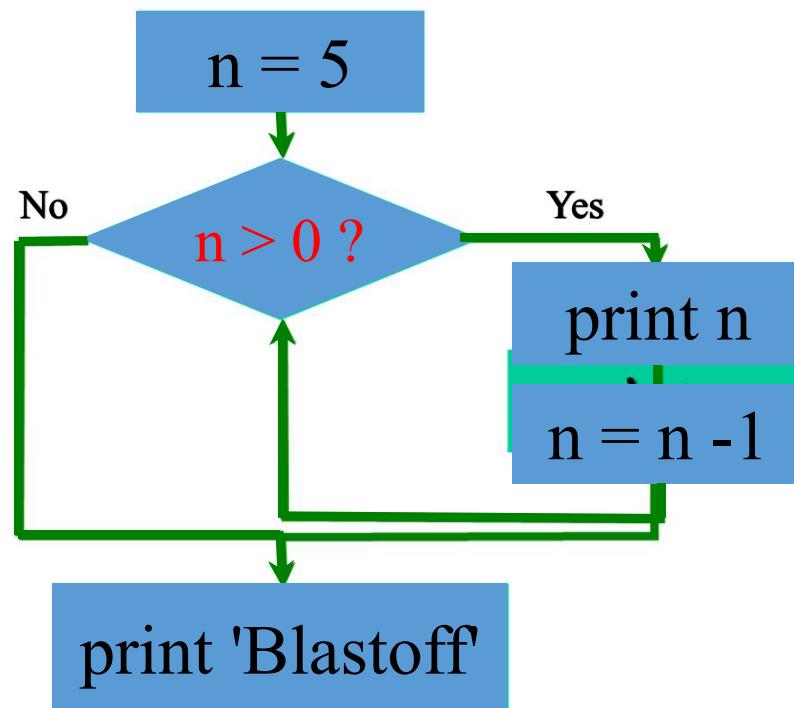
## Program steps or flow

- Like a recipe or installation instructions, a program is a **sequence** of steps to be done in order.
- Some steps are **conditional** - they may be skipped.
- **Sometimes a step or group of steps are to be repeated.**
- Sometimes we store a set of steps to be used over and over as needed several places throughout the program. We call it **function**.

# Loop statement

- Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.
- We call repeated execution of a set of programming statements, the **loop** statement.
- In Python, we have **while** and **for** loop.

## Count-down example

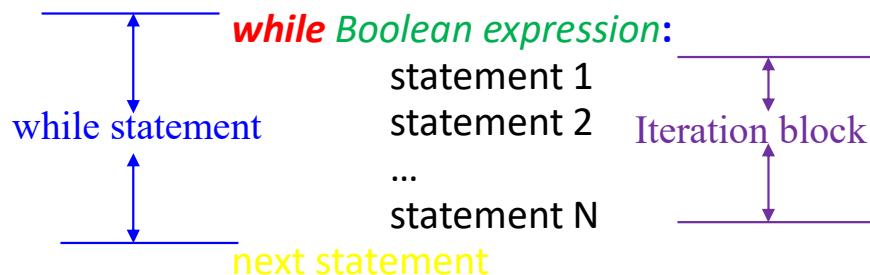


Output:

5  
4  
3  
2  
1  
Blastoff!

# The **while** statement

- Syntax:



- The flow of execution

1. Evaluate the `expression`, yielding Boolean value `True` or `False`
2. If the expression is `False`, exit the entire while statement and continue execution at the `next statement`
3. If the expression is `True`, execute each of the statements in the body and then go back to step (1)

## Example

```
x = 5
while x > 3:
    print(x)
    x = x - 1
print(x+1)
```

### Iteration 3

```
x = 3
x > 3 (False) # exit the
iteration
print(x+1)      # update x
#start
```

```
5
4
4
```

## Example

```
x = 5
while x > 3:
    print(x)
    x = x - 1
print(x+1)
```

Iteration 1

x = 5

Iteration 2

x = 4

Iteration 3

x = 3

x > 3 (False) # exit the  
iteration

print(x+1) # update

x

#start

5  
4  
4

## Count-Controlled Loops

- A **while** loop that is controlled by a counter.
  - Used when you know how many iterations it will take to exit the loop but don't know what may happen then.

```
# Initialize the counter
counter = 1
# Check the counter
while counter <= 10 :
    print(counter)
    # Update the counter
    counter = counter + 1
```

```
# Initialize the counter
counter = 10
# Check the counter
while counter >= 0 :
    print(counter)
    # Update the counter
    counter = counter - 1
```

## Definite loop

- Quite often we have a list of items – effectively a finite set of things (for example: counter)
- These loops are called “definite loops” because they execute an exact number of times

## Event-Controlled Loops

- A **while** loop that is controlled by an event.
  - Used when you know what will happen (event) to exit a loop but don't know how many iterations it may take.

```
# Initialize the loop variable
balance = 1000
year = 1
# Check the loop variable
while balance <= 2000:
    year = year + 1
    # Update the loop variable
    balance = balance * 1.2
print(year)
```

## Indefinite loop

- **While** loops are called “**indefinite loops**” because they keep going until a Boolean expression becomes **False**
- The loops we have seen so far are easy to examine to see if they will terminate or if they are “infinite loops”
- Sometimes it is harder to be sure if a loop will terminate

## Example

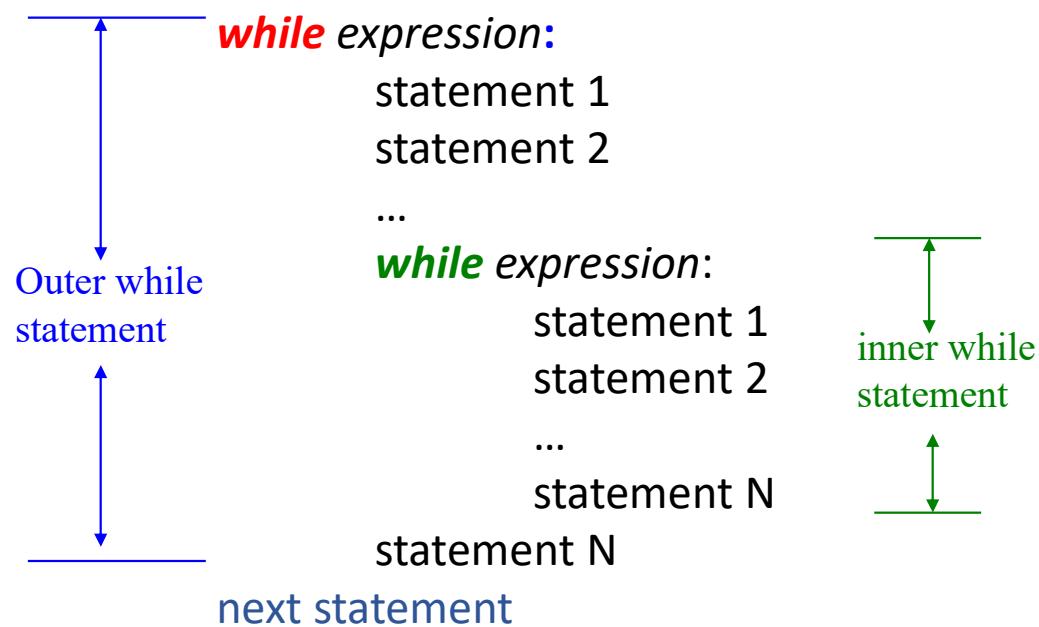
- Work out the result of the following script.

```
x = 5
while x != 1:
    print (x)
    if x%2 == 0:
        x = x / 2
    else:
        x = x*3 + 1
print ("Finished")
```

```
5
16
8
4
2
Finished
```

## The nested ***while*** statement

- Syntax:



## Example

- Work out the result of the following script.

```
x = 5
while x != 1:
    print(x)
    while x > 3:
        print("x>3")
        x = x - 1
    x = x - 1
```

```
5
x>3
x>3
2
```

## Breaking out of a loop

- The `break` statement ends the **current innermost loop** and **jumps** to the statement **immediately following** that loop.
- It can happen anywhere in the body of the loop, depending on your needs.

```
while True:
```

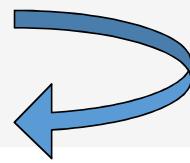
```
    line = input('> ')
```

```
    if line == 'done' :
```

```
        break
```

```
        print (line)
```

```
    print ("Done!")
```



```
> hello there
```

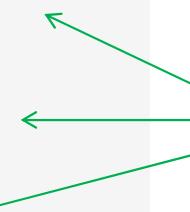
```
hello there
```

```
> finished
```

```
finished
```

```
> done
```

```
Done!
```

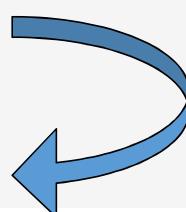


Texts in green here are  
received from the  
keyboard

## Breaking out of a loop

- All statements in the loop body, even statements after `break`, will **NOT** be executed if break happens.

```
x = 5
while x > 0:
    print (x)
    if x == 3:
        break
        x = x - 1
        print(x)
    print (x)
```

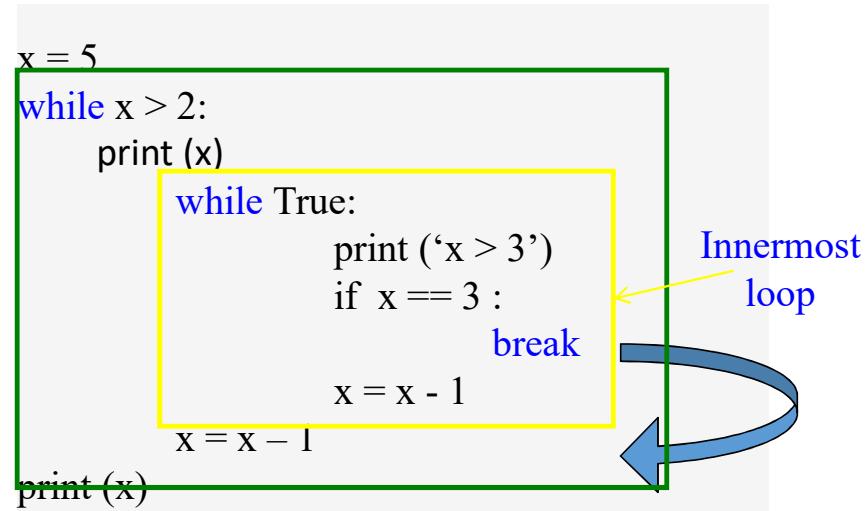


Output:

```
5
4
4
3
3
3
```

## Breaking out of a loop

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



Output:

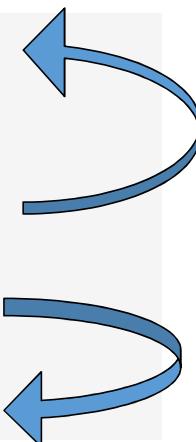
5  
x>3  
x>3  
x>3  
2

## The `continue` statement

Same as `next` in R.

- The `continue` statement ends the current iteration of the innermost loop and returns to the top of the loop and starts the next iteration.
- It can happen anywhere in the body of the loop, depending on your needs.

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



```
> hello there  
hello there  
> #  
> print this!  
print this!  
> done  
Done!
```

Texts in green here  
are received from  
the keyboard

# Example

```
x = 5
while x > 0:
    x = x - 1
    if x == 3:
        continue
    print(x)
print(x)
```

Output:

4  
2  
1  
0  
0

## Example

```
x = 5
while x > 2:
    print(x)
    while x > 0:
        x = x - 1
        if x < 3 :
            continue
            print ('x < 3')
        else:
            print ('x >= 3')
    x = x - 1
print(x)
```

Innermost  
loop

Output:

```
5
x>=3
x>=3
-1
```

## Summary of the `while` Loop

- `while` loops are very common
- Initialize variables before you test
  - The condition is tested **BEFORE** the loop body
    - This is called pre-test
    - The condition often uses a counter variable
  - Something inside the loop should (not always) change one of the variables used in the test
- Watch out for infinite loops!

# The **for** statement

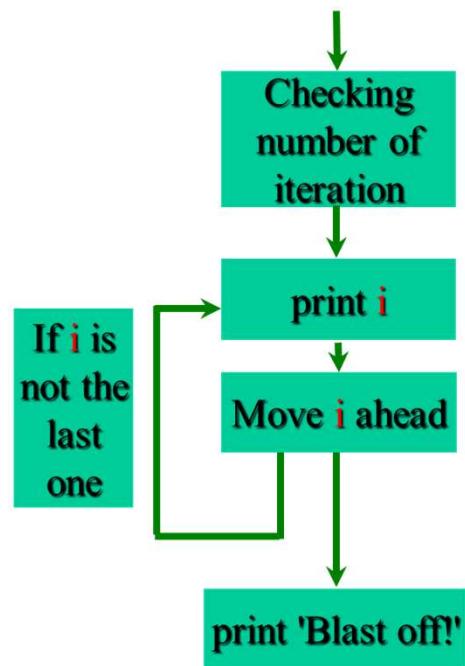
- Syntax:

```
for iterator in expression_list:  
    statement 1  
    statement 2  
    ...  
    statement N
```

- The flow of execution

1. The expression list is only evaluated **once**; it should yield an iterable object, a sequence of items (e.g., list, tuple, etc.)
2. For each member or item in the expression\_list, execute **all** statements in the for body.

# The *for* statement



```
for i in [5, 4, 3, 2, 1]:  
    print (i)  
    print ('Blastoff!')
```

```
5  
4  
3  
2  
1  
Blastoff!
```

## The **for** statement

- **for** loop is definite loop with explicit **iteration variable** that changes each time.
- The **iteration variable** “iterates” through the **sequence** (ordered set)
- The **block (body)** of code is executed once for each element **in** the **sequence**
- The **iteration variable** moves through all of the values **in** the **sequence**

The diagram illustrates a Python for loop with three annotations:

- A green arrow labeled "Iteration variable" points to the variable `i` in `for i in [5, 4, 3, 2, 1]:`.
- An orange arrow labeled "Five-element sequence" points to the list `[5, 4, 3, 2, 1]`.
- A pink arrow labeled "block (body)" points to the code block `print(i)`.

```
for i in [5, 4, 3, 2, 1]:  
    print(i)
```

## Example

```
for i in [5, 4, 3, 2, 1]:  
    if i % 2 == 0:  
        print (i, ": even")  
    else:  
        print (i, ": odd")  
print ("Blastoff!")
```

```
5: odd  
4: even  
3: odd  
2: even  
1: odd  
Blastoff!
```

## Nested **for** statement

- Syntax:

The diagram illustrates the syntax of nested for loops. It features two vertical double-headed arrows. The left arrow, colored blue, spans the entire height of the outer loop's code block. The right arrow, colored green, spans the entire height of the inner loop's code block. The outer loop's code block starts with a blue **for** iterator and ends with a blue **statement (outer for)**. The inner loop's code block starts with a green **for** iterator and ends with a green **statement (inner for)**.

```
for iterator in expression_list:  
    statement 1  
    statement 2  
    ...  
    for iterator in expression_list:  
        statement 1  
        ...  
        statement N  
    statement (outer for)  
statements (after outer for)
```

## Nested **for** statement

- The flow of execution
  1. Consider the “inner for loop” as “one statement” within the outer loop body
  2. For each member in the “outer loop”, execute all statements
  3. When execute inner for loop statement, consider it as a real loop

## Example (1)

```
for i in [1, 2, 3]:  
    for j in [1, 2, 3]:  
        print (i*j)  
print ('Done')
```

Outer iteration 1

i = 1

Outer iteration 2

i = 2

Outer iteration 3

i = 3

for j in [1, 2, 3]:  
 print (i\*j)

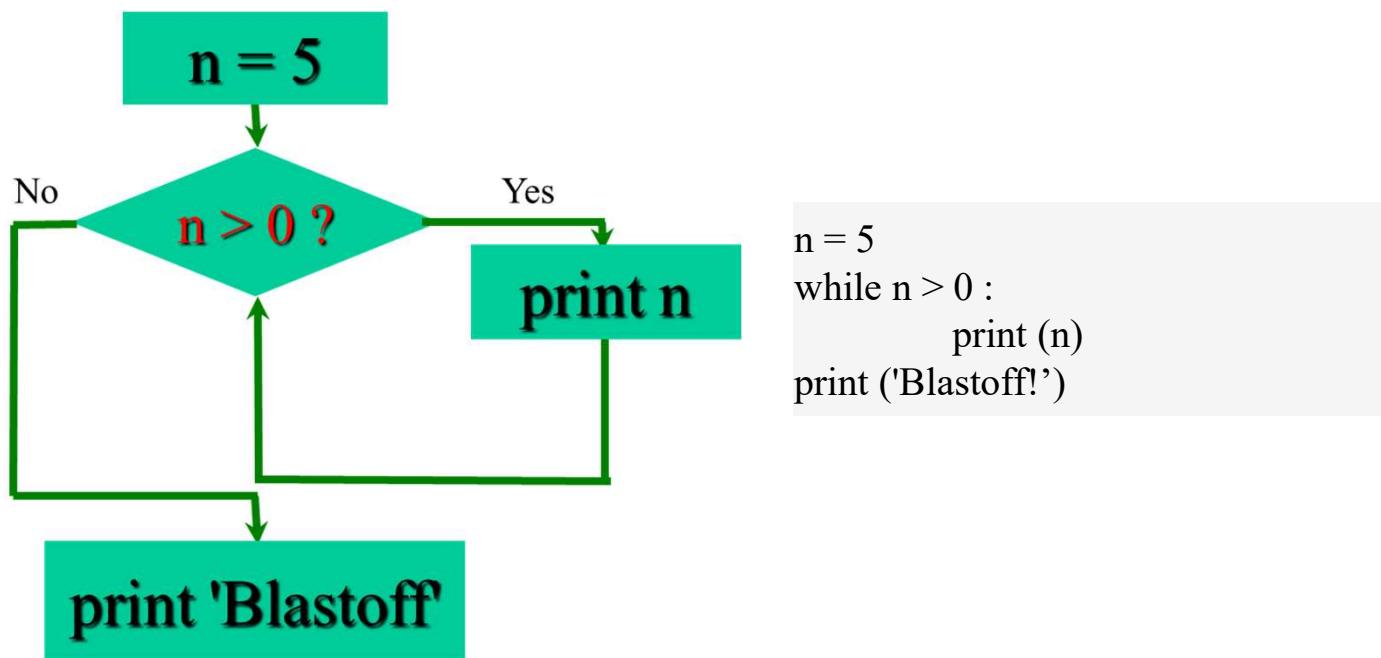
1  
2  
3  
2  
4  
6  
3  
6  
9  
Done

## Example (2)

```
for i in [1, 2, 3]:  
    j = 1  
    while j<=i:  
        print (i)  
        j = j+1  
print ('Done')
```

Output
1
2
2
3
3
3
Done

## What is wrong with this loop?

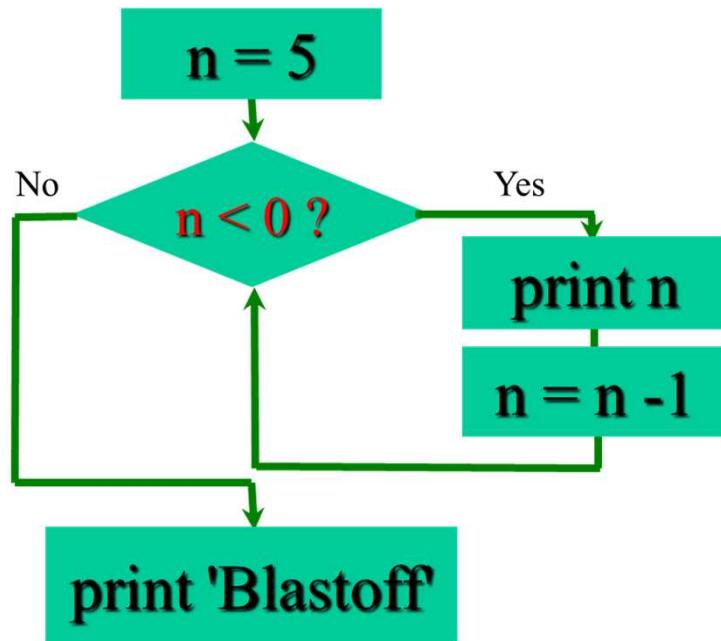


## Common Error 1: Infinite Loops

- The loop body will execute until the test condition becomes **False**.
- What if you forget to **update** the test variable?
  - n is the test variable (n doesn't change)
  - You will loop forever! (or until you stop the program)

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

## What is wrong with this loop?



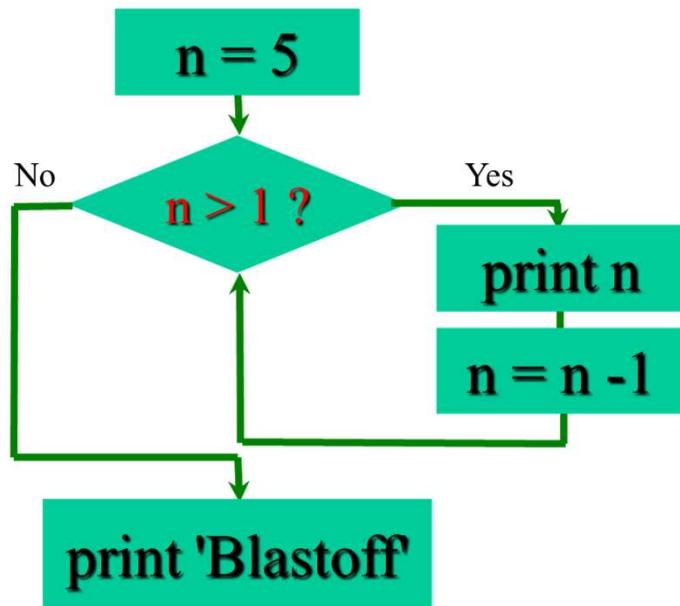
```
n = 5  
while n < 0 :  
    print (n)  
    n = n - 1  
print ('Blastoff!')
```

## Common Error 2: Incorrect Test Condition

- The loop body will only execute if the test condition is **True**.
- What if you give a incorrect test condition?

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

## What is wrong with this loop?



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

## Common Error 3: Off-by-One Errors

- A ‘counter’ variable is often used in the test condition.
- Your counter can start at 0 or 1, but programmers often start a counter at 0
- If I want to paint all 5 fingers on one hand, when I am done?
  - If you start at 0, use “`<`”                      If you start at 1, use “`<=`”

```
1 2 3 4 5
finger = 1
FINGERS = 5
while finger <= FINGERS :
    # paint finger
    finger = finger + 1
```

```
1 2 3 4 5
finger = 0
FINGERS = 5
while finger < FINGERS :
    # paint finger
    finger = finger + 1
```

## while or for

- while loop must **test a condition** before each iteration, loops until a condition is **False**.
- for loop does **Not** test a condition, instead it simply iterates through a sequence (a collection or iterable object or generator function)
- for loop may be seen as a simpler version of **counter-controlled while** loop.
  - ✓ No need to create artificial counter.
  - ✓ No need to test the condition.

# Example

```
counter = 1
while counter <= 10 :
    print(counter)
    counter = counter + 1
```

```
for counter in range (1,11):
    print(counter)
```

# Exercise

- Write a program to find out all the leap years in a time period (for example, 2001 to 2100), and print out these years and total number of leap year. You can use condition statement from earlier exercise to check if a given year is a leap year.

# Exercise

- Write a program that asks the user to enter a positive integer n, and prints out the following shape. You must use nested loop in your program. The gap between two numbers could be a space or a tab.
- Sample Run (suppose the n here is 5)
  - 1
  - 1 2
  - 1 2 3
  - 1 2 3 4
  - 1 2 3 4 5

# More data types

*Everything is an object in Python programming, data types are actually classes and variables are instance (object)*

- **Number**
- **String**
- **List**
- **Tuple**
- **Dictionary**

We don't have Vectors, Matrices, Arrays, Factors, Data Frames as built-in data type. We have other library to handle more complex data type.

# Sequence Data

- Strings, lists, and tuples are all sequence types, so called because they behave like a sequence - an ordered collection of objects.
- Sequence types are qualitatively different from numeric types because they are compound data types - meaning they are made up of smaller pieces.
- Sequence types share a lot in common but with significant differences.

# List

## List

- A list is a kind of collection.
- A collection allows us to put many values in a single “variable”.
- You create a list with square bracket [].
  - ❑ scores = [50, 60, 90]
  - ❑ friends = ['alice', 'bob', 'charle']
  - ❑ Scores = []

## List

- List is surrounded by square brackets and the elements in the list are separated by commas.
- A list element can be **any** Python object – even another list.
- A list can be empty

```
>>> print([1,23,45])
[1,23,45]
>>> print(['red','blue'])
['red','blue']
>>> print([25,'green'])
[25,'green']
>>> print([1,[3,4],37])
[1,[3,4],37]
>>> print([])
[]
```

138

## Accessing elements: indexing

- We can get at any single element in a list using an index specified in **square brackets**.
- Index starts from **0**.
- Index must be integer.

Index in R starts with 1 while  
starts with 0 in Python and  
many other languages.

1	3	45	10
Index: 0	1	2	3

```
>>> x = [1,3,45,10]
>>> print (x[1])
3
```

139

## Accessing elements : indexing

- If the index is negative value, it counts backward from the end of the list.

1	3	45	10
Index:	-4	-3	-2

>>> x = [1,3,45,10]

>>> print (x[-1])

10

## Lists are **mutable**

- Lists are “**mutable**” – we can change an element of a list using index operator.

1	3	45	10
Index:	0	1	2
			3
1	3	33	10
Index:	0	1	2
			3

```
>>> x = [1,3,45,10]  
>>> x[2] = 33  
>>> print (x)  
[1, 3, 33, 10]
```

## List length: `len()`

- The `len()` function takes a list as a parameter and returns the number of elements in the list
- `numbers1 = [1, 3, 45, 10]`
- `numbers2 = [1, [3, 45], 10]`

```
>>> numbers1 = [1,3,45,10]
>>> print (len(numbers1))
4
>>> numbers2 = [1,[3,45],10]
>>> print (len(numbers2))
3
```

## The *range* function

- The `range(x)` function returns a list of numbers that range from zero to  $x-1$
- The `range(x, y, [z])` function returns a list of numbers that range from  $x$  to  $y-1$  [at step of  $z$ ].
- If  $x>y$ , returns an empty list

## The *range* Function

1. class **range(stop)**
2. class **range(start, stop[, step])**
  1. Start: The value of the start parameter (or 0 if the parameter was not supplied)
  2. Stop: The value of the stop parameter
  3. Step: The value of the step parameter (or 1 if the parameter was not supplied)

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

## List membership

- `in` is a Boolean operator that tests membership in a sequence.
- `not in` to test whether an element is not a member of a list
- They do not modify the list

```
>>> fruit = ['apple', 'banana', 'orange']
>>> 'banana' in fruit
True
```

```
>>> x = [3,4,5,6,7,8]
>>> 2 not in x
True
```

## List operations (1): Concatenation

- We can create a new list by **concatenating** two existing lists together with **+** sign.
  - A new list that contains the elements of the first list, followed by the elements of the second

```
>>> a = [1,2,3]
>>> b = [4,5,6]
>>> c = a + b
>>> print (c)
[1,2,3,4,5,6]
[[1,2,3],[4,5,6]]
```

```
>>> d = ['x','y']
>>> e = a + d
>>> print (e)
[1,2,3,'x','y']
```

## List operations (2): Replication

- We can create a new list by **replicating** an existing list with **\*** sign.

```
>>> a = [1,2,3]
>>> b = a * 3 # same as 3 * a
>>> print (b)
[1,2,3,1,2,3,1,2,3]
```

## List operations (3): slice

- Lists can be **sliced** using :
  - ❑ `ListName[x:y]` returns a sublist from index `x` to index `y-1`
  - ❑ `ListName[:x]` returns a sublist from index `0` to index `x-1`
  - ❑ `ListName[x:]` returns a sublist from index `x` to the end

```
>>> a = [9,41,12,3,77,19]
>>> a[1:3]
[41,12]

>>> a[:4]
[9,41,12,3]

>>> a[3:]
[3,77,19]
```

148

## List operations (4): delete

- Using slices : to delete list elements is error prone.
- Python provides an alternative **del** that more readable
  - ❑ **del** listName[i] delete the element with index i
  - ❑ **del** listName[i:j] delete elements with index from i to j-1

```
>>> a = [9,41,12,3,77,19]
>>> a[1:3] = []
>>> print(a)
[9,3,77,19]

>>> del a[1]
>>> print(a)
[9,77,19]

>>> del a[:2]
>>> print (a)
[19]
```

## List methods (1): **append()**

- A **method** in Python is an object-specific function. It can only be called by a name that is associated with an object.
- Building a list via appending
  - We can create an **empty list** and then add elements using the **append** method:  
`append(new_element)`
  - The list stays in order and new elements are added at the **end** of the list.

```
>>> a = list() #a = []
>>> a.append('book')
>>> a.append(30)
>>> print (a)
['book', 30]
```

150

## List methods (2): sort()

- A list is an ordered sequence, so that
  - ❑ A list can be sorted (i.e., change its order)
  - ❑ The `sort` method means “*sort yourself*”

```
>>> a = ['Joseph', 'Glenn', 'Sally']
>>> a.sort()
>>> print(a)
['Glenn', 'Joseph', 'Sally']

>>> print(a[1])
Joseph
```

- More method: <https://docs.python.org/3/tutorial/datastructures.html>

## List Aliasing

- When you “copy” a list variable into another, both variables refer to the same list
  - The second variable is an *alias* for the first because both variables reference the same list

```
scores = [10, 9, 7, 4, 5]
values = scores # Copying list reference
```

A list variable specifies the location of a list. Copying the reference yields a second reference to the same list.

scores = \_\_\_\_\_  
values = \_\_\_\_\_

List contents

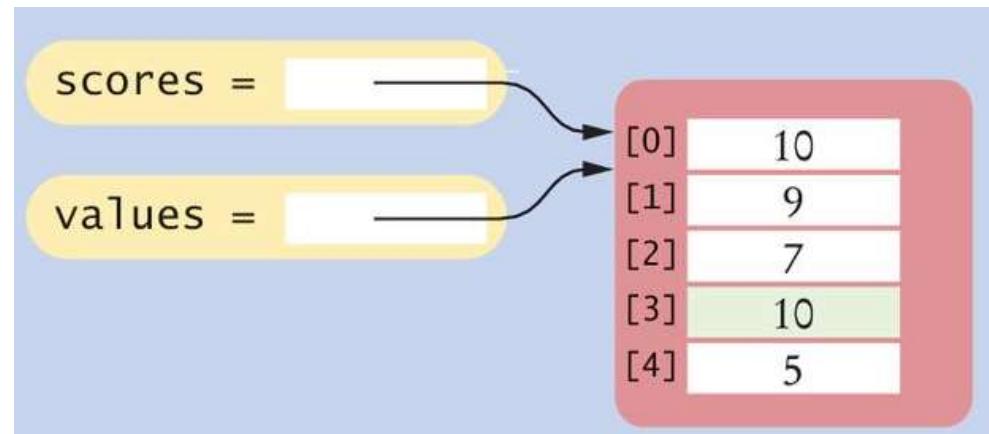
[0]	10
[1]	9
[2]	7
[3]	4
[4]	5

References

## Modify Aliased Lists

- You can **modify** the list through **either** of the variables:

```
scores[3] = 10  
print(values[3])    # Prints 10
```



## Copy a list

- You can use the built in list() function:
  - new\_list = list(old\_list)
- Essentially, you are creating a new list variable from scratch.

```
scores = [10, 9, 7, 4, 5]
values = list(scores)      # Copying list
scores[3] = 10
print(values[3])    # Prints 4
```

## Lists are comparable

- The comparison operators work with list.
- If the first item is equal, Python goes on to the next element, and so on, until it finds elements that differ.
- Comparing elements must be comparable.

```
>>> [0, 1, 2] < [5, 1, 2]
True
>>> [0, 1, 2] < [0, 1, 3]
True
>>> ["abc", "def"] < ["acd", "def"]
True
>>> ["abc", "def"] < ["abc", "deg"]
True
>>> [1, 2, "b"] < [1, 3, "a"]
True
>>> [1, 2, "b"] < [1, "a", "a"]
TypeError: '<' not supported between instances of 'int' and 'str'
```

## Some Built-in functions

- **max( )** returns the largest number or the letter. (will not work when numbers and letters are mixed)
- **min( )** returns the smallest number or the letter. (will not work when numbers and letters are mixed)
- **sum( )** returns the sum of all the numbers. (will only work with numbers only)

```
>>> a = [3,44,13,11,77,15]
>>> print max(a)
77
>>> print min(a)
3
>>> print sum(a)
163
>>> a = ["a", "b", "c"]
>>> print max(a)
c
>>> print min(a)
a
>>> print sum(a)
TypeError: unsupported
operand type(s) for +:
'int' and 'str'
```

# Example

```
numList = list()

while True:
    input = input('Enter a number: ')
    if input == 'done':
        break
    value = float(input)
    numList.append(value)

average = sum(numList) / len(numList)
print ('Average: ',average)
```

# Matrices

- Nested lists are often used to represent matrices.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
>>> matrix = [[1,2,3],[4,5,6],[7,8,9]]
```

```
>>> matrix[1]
```

```
[4,5,6]
```

```
>>> matrix[1][2]
```

```
6
```

# Out of Range Errors

- Perhaps the most common error in using lists is accessing a non-existent element
- If your program accesses a list through an out-of-range index, the program will generate an exception.

```
values = [2.3, 4.5, 7.2, 1.0, 12.2, 9.0, 15.2, 0.5]
values[8] = 5.4
# Error--values has 8 elements,
# and the index can range from 0 to 7
```

# **Tuple**

160

## Tuples

- A tuple is similar to a list, but once created, its contents cannot be modified (a tuple is an **immutable** version of a list).
- A tuple is created by specifying its contents as a comma-separated sequence. You can enclose the sequence in parentheses:
  - If you prefer, you can omit the parentheses:

```
>>>triple = (5, 10, 15)
>>>triple = 5, 10, 15
>>>print(triple)
(5, 10, 15)
```

## Tuples are more efficient

- Since Python does not have to build tuple structures to be modifiable, they are simpler and more efficient in terms of memory use and performance than lists
- So in our program when we are making “**temporary variables**”, we prefer tuple over lists

# Tuple

- To create a tuple with a single element, we have to include the final comma
  - `tuple = ('a',)`

```
>>> t1 = ("a")
>>> t2 = ("a",)
>>> type(t1)
str
>>> type(t2)
tuple
```

## Tuple operations: indexing and slicing

- Similar to lists, tuples can also be **Indexed** and **sliced**.
  - Use [] to specify the position, the same as list operation.

1	3	45	10
Index:	0	1	2

```
>>> x = (1,3,45,10)
>>> x[2]
45
>>> x[1:3]
(3,45)
>>> x[:3]
(1,3,45)
>>> x[3:]
(10,)
```

164

## Tuple membership

- You can also test if an element is `in` or `not in` a tuple the same way as list.

```
>>> x = (3,4,5,6,7,8)
>>> 3 in x
True
>>> 4 not in x
False
```

## “modify” a tuple

- However, a tuple **cannot** be modified (tuple is **immutable**).

```
>>> triple = (5, 10, 15)
>>> triple[1] = 2
TypeError: 'tuple' object does not
support item assignment
```

- Even we can not modify the elements of a tuple, we can replace it with a different tuple.

```
>>> tuple = (1,) + triple[1:]
>>> print(tuple)
(1, 10, 15)
```

## Tuple operations: Concatenation and Replication

- Tuple also has concatenation and replication operations.

```
>>> a = (1,2,3)
>>> b = (4,5,6)
>>> a + b
(1,2,3,4,5,6)
>>> a * 3
(1,2,3,1,2,3,1,2,3)
```

## Tuples and assignment

- We can put a tuple on the left hand side of an assignment statement
- We can even omit the parenthesis
- To swap two values, we can use tuple assignment to neatly solve this problem

```
>>> (x, y) = (4, 'hello')
>>> print (y)
hello
```

```
>>> a, b = (1,7)
>>> a, b = b, a
>>> print (a)
7
```

## Tuples are **comparable**

- The comparison operators also work with tuples in the similar way as lists.
- However, you cannot compare a tuple with a list.

```
>>>(0, 1, 2) < (5, 1, 2)
True
>>>("abc", "def") < ("acd", "def")
True
>>> (1, 2, "b") < (1, 3, "a")
True
>>> (1, 2, "b") < (1, "a", "a")
TypeError: '<' not supported
between instances of 'int' and
'str'
>>>(0, 1, 2) < [5, 1, 2]
TypeError: '<' not supported
between instances of 'list' and
'tuple'
```

## Tuple method

- Tuples have **NO** method.

```
>>>triple = (5, 10, 15)
>>>triple.append(20)
AttributeError: 'tuple' object has no
attribute 'append'
>>>triple.sort()
AttributeError: 'tuple' object has no
attribute 'sort'
```

# **String**

## The string data type

- Text is represented in programs by the *string* data type.
- A string is a sequence of characters enclosed within double quotation marks ("") or single quotation marks (')
- Like tuple, string is also **immutable**.

```
>>> str1 = "Hello"  
>>> str2 = 'spam'  
>>> print (str1, str2)  
Hello spam  
>>> type(str1)  
<class 'str'>  
>>> type(str2)  
<class 'str'>
```

## String operations: indexing and slicing

- Similar to list and tuple, we can access the individual characters in a string through *indexing*.
- We can also *slice* a string.

H	e	I	I	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet = "Hello Bob"  
>>> greet[0]  
'H'  
>>> print(greet[0], greet[2], greet[4])  
H l o  
>>> x = 8  
>>> print(greet[x - 2])  

```

174

## String operations: indexing and slicing

- Similar to list and tuple, we can access the individual characters in a string through *indexing*.
- We can also *slice* a string.

H	e	I	I	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet[0:3]  
'Hel'  
>>> greet[5:9]  
' Bob'  
>>> greet[:5]  
'Hello'  
>>> greet[5:]  
' Bob'  
>>> greet[:]  
'Hello Bob'
```

## The string operations

- *Concatenation* “glues” two strings together (+)
- *Replication* builds up a string by multiple concatenations of a string with itself (\*)

```
>>> "spam" + "eggs"  
'spameggs'  
>>> "Spam" + "And" + "Eggs"  
'SpamAndEggs'  
>>> 3 * "spam"  
'spamspamspam'  
>>> "spam" * 5  
'spamspamspamspamspam'  
>>> (3 * "spam") + ("eggs" * 5)  
'spamspamspameggseggseggsseggs'
```

## String methods

- `s.capitalize()` – Copy of s with only the first character capitalized
- `s.title()` – Copy of s; first character of each word capitalized
- `s.upper()` – Copy of s; all characters converted to uppercase
- `s.lower()` – Copy of s in all lowercase letters
- `s.split()` – Split s into a list of substrings

## String methods

- `s.find(sub)` – Find the first position where sub occurs in s
- `s.replace(oldsub, newsub)` – Replace occurrences of oldsub in s with newsub
- `s.count(sub)` – Count the number of occurrences of sub in s
- Many more can be found:  
<https://docs.python.org/3.4/library/stdtypes.html#string-methods>

## List, Tuple and String

- They are all sequence types, meaning the elements are ordered.
- **Most** operations for sequence data can be applied to all three: index, slice, membership, concatenation, replication, comparison, etc.
- Methods are object-specific, meaning each data type may have its own set of methods.
- List is mutable while tuple and string are immutable.

## Iterate a List

- Given the list that contains 10 elements, we will want to set a variable, say *i*, to 0, 1, 2, and so on, up to 9

```
# First version (list index used)
for i in range(10) :
    print(i, values[i])
```

```
# Better version (list index used)
for i in range(len(values)) :
    print(i, values[i])
```

## Iterate a List

```
# Third version: index values not needed (traverse  
# list elements in order)  
for element in values :  
    print(element)
```

```
fruits = ["Apple", "Banana", "Orange"]  
for fruit in fruits: #fruit is a new variable  
    print(fruit)  
#output:  
Apple  
Banana  
Orange
```

# Dictionaries

182

## Dictionaries

- The compound types we have learned: lists and tuples – are sequence types that use integers as indices.
- Dictionaries are similar to these type except that they can use other data type as an index.
- Create an empty dictionary with braces {} or dict().

❑ eng2sp = {}

❑ eng2sp = dict()

```
>>> eng2sp = {}
>>> eng2sp['one'] = 'uno'
>>> eng2sp['two'] = 'dos'
>>> print eng2sp
{'one': 'uno', 'two': 'dos'}
```

## Dictionaries

- Dictionaries are like bags – no order (index)

```
>>> purse = dict()
>>> purse['money'] = 12
>>> purse['candy'] = 3
>>> purse['tissues'] = 75
>>> print purse
{'money': 12, 'tissues': 75, 'candy': 3}
>>> print purse['candy']
3
>>> purse['candy'] = purse['candy'] + 2
>>> print purse
{'money': 12, 'tissues': 75, 'candy': 5}
```

## 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(180)  
>>> print lst  
[21,180]
```

```
>>> lst[0] = 23  
>>> print lst  
[23,180]
```

```
>>> ddd = dict()  
>>> ddd['age'] = 21  
>>> ddd['score'] = 90  
>>> print ddd  
{'score':90, 'age':21}
```

```
>>> ddd['age'] = 23  
>>> print ddd  
{'score':90, 'age':23}
```

## Dictionary operations

- **del** statement removes a key-value pair from a dictionary
- We can also change the value associated with a key
- It is an error to reference a key which is not in the dictionary

```
>>> ddd = dict()  
>>> ddd['age'] = 21  
>>> ddd['score'] = 90  
>>> print(ddd)  
{'age':21, 'score':90}  
  
>>> del ddd['age']  
>>> print(ddd)  
{'score':90}  
  
>>> print(ddd['height'])  
KeyError: 'height'
```

## Dictionary methods

- `dictName.keys()` returns a **list** of the keys.
- `dictName.values()` returns a **list** of the values in the dictionary
- `dictName.items()` returns both, in the form of a **list** of **tuples** – one for each key-value pair

```
>>> ddd = {'age':21,'score':90}  
>>> ddd.keys()  
['age', 'score']
```

```
>>> ddd.values()  
[21, 90]
```

```
>>> ddd.items()  
[('age', 21), ('score', 90)]
```

## Dictionary methods

- We can also use the `in` operator to see if a key is in the dictionary
- To see if a value in the dictionary, use the method `values()`

```
>>> ddd = {'age':21,'score':90}
>>> 'age' in ddd
True
>>> 21 in ddd
False
>>> 21 in ddd.values()
True
```

## Dictionary methods

- `dictName.get(key)` returns the value if the key appears in the dictionary
- `dictName.get(key, 0)` returns the value if the key appears in the dictionary, 0 otherwise

```
counts = dict()
names = ['csev', 'cwen', 'csev', 'zqian', 'cwen']
for name in names:
    counts[name] = counts.get(name,0) + 1
print (counts)
```

## Iterate dictionaries

```
counts = {'chuck' : 1, 'fred' : 42, 'jan' : 100}
for key in counts:
    print (key, counts[key])
```

```
jan 100
chuck 1
fred 42
```

## Two iteration variables

- We loop through the key-value pairs in a dictionary using two iteration variables
- Each iteration, the first variable is the key and the second variable is the corresponding value for the key

```
students = {'name':'alice', 'age':20, 'gender': 'f'}
for k,v in students.items():
    print (k,":",v)
```

Outputs:

```
name: alice
age: 20
gender: f
```

191

## Exercise

- Write a program to find out all the leap years in a time period (for example, 2001 to 2100), and store your results in a list. You can reuse script from yesterday's exercise to identify all the leap years in a time period.

## Exercise

- You receive one sentence and one specific delimiter from the keyboard. Then you split the sentence into strings that are divided by the delimiter and store your result as a list. For example, sentence: “I like computer programming, including Python, Java, and C/C++”  
delimiter: , (comma)

Then the output list will be:

[‘I like programming’, ‘including Python’, ‘Java’, ‘and C/C++’].

If your delimiter is “/”, then your output list will be [‘I like programming, including Python, Java, and C’, ‘C++’]

Note: you may use method like s.find() and indexing operation to help you separate the string.

# Functions

## Reusable codes

### Calc:

Do the following operations for two numbers and print results:  
+, -, \*, /

```
def calc(a, b):
    sum = a + b
    subtraction = a - b
    multiply = a * b
    division = a / b
    print ("sum is: ", sum)
    print ("subtraction is: ", subtraction)
    print ("multiplication is: ", multiply)
    print ("division is: ", division)
```

```
>>> calc(5,7)
>>> calc(3, 4)
>>> calc (1.2, 2.5)
```

## Function

- A function is a block of **organized, reusable code** that is used to perform a group of related actions.
- Functions provide better **modularity** for your application and a high degree of code reusing.

# Python functions

- There are two kinds of functions in Python
  - ❑ Build-in functions that are provided as part of Python
    - `input()`, `type()`, `print()`, `int()`, ...
  - ❑ Functions that we define ourselves and then use

# Build-in functions

- Math functions
  - ❑ Python has a math module that provides most of the familiar mathematical functions.
  - ❑ Before use all these functions, we have to import them:
    - `>>> import math`
  - ❑ To call one of the functions, we have to specify the name of the module and the name of the function, separated by a dot. This format is also called **dot notation**.
    - ❑ `>>>decibel = math.log10(17)`
    - ❑ `>>>angle = 1.5`
    - ❑ `>>>height = math.sin(angle)`

## Math function examples

- `>>> degrees = 45`
- `>>> angle = degrees * 2 * math.pi / 360`
- `>>> math.sin(angle)`
- `0.707106781187`
- Combination of functions
- `>>> x = math.exp(math.log(10))`

# Many other math functions

Function name	Description
<code>abs (value)</code>	absolute value
<code>ceil (value)</code>	rounds up
<code>cos (value)</code>	cosine, in radians
<code>degrees (value)</code>	convert radians to degrees
<code>floor (value)</code>	rounds down
<code>log (value, base)</code>	logarithm in any base
<code>log10 (value)</code>	logarithm, base 10
<code>max (value1, value2, ...)</code>	larger of two (or more) values
<code>min (value1, value2, ...)</code>	smaller of two (or more) values
<code>radians (value)</code>	convert degrees to radians
<code>round (value)</code>	nearest whole number
<code>sin (value)</code>	sine, in radians
<code>sqrt (value)</code>	square root
<code>tan (value)</code>	tangent

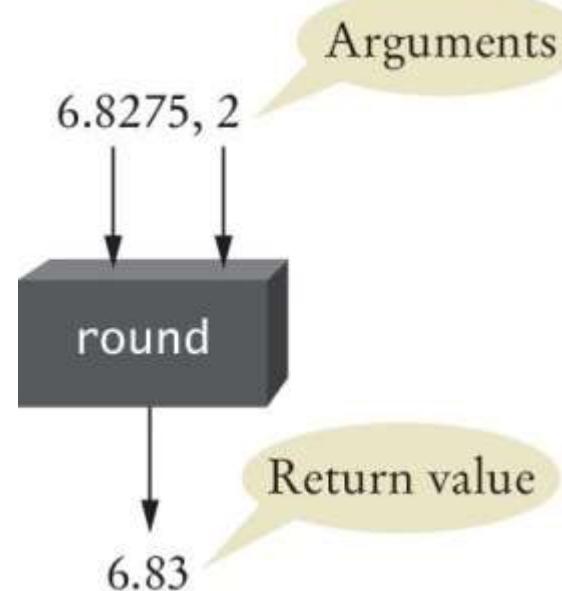
Constant	Description
e	2.7182818...
pi	3.1415926...

## Function as a Black Box

- A thermostat is a ‘black box’
  - Set a desired temperature
  - Turns on heater/AC as required
  - You don’t have to know how it really works!
    - How does it know the current temp?
    - What signals/commands does it send to the heater or A/C?
- Use functions like ‘black boxes’
  - Pass the function what it needs to do its job
  - Receive the answer

## The `round()` Function as a Black Box

- You pass the `round()` function its necessary input arguments (`6.8275` & `2`) and it produces its output result (`6.83`)



## The `round()` Function as a Black Box

- You may wonder how the round function performs its job
- As a **user** of the function, you don't need to know how the function is implemented
- You just need to know the specification of the function:
  - If you provide input  $x$  and  $n$ , the function outputs  $x$  rounded to  $n$  decimal digits

## Designing Your Own Functions

- When you design your own functions, you will want to make them appear as black boxes to other programmers.
  - Even if you are the only person working on a program, making each function into a black box pays off: there are fewer details that you need to keep in mind

## Defining our own functions

- Simple rules to define a function
  1. Function blocks begin with the keyword **def** followed by the function name, parentheses (), and a colon :
  2. Any **parameters** should be placed within these parentheses.
  3. The code block within every function is **indented**.

```
def functionname (parameters):
```

```
    statement 1  
    statement 2  
    ...  
    statement n
```

## Building our own functions

- Defining a function doesn't mean we **execute** the body of the function

```
x = 5
print ('Hello')

def print_oks():
    print ('OK 1')
    print ('OK 2')

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

```
Hello
Yo
7
```

## Programming Tip: Function Comments

- Whenever you write a function, you should *comment* its behavior
- Remember, comments are for human readers, not compilers

```
## Computes the volume of a cube.  
# @param sideLength the length of a side of the cube  
# @return the volume of the cube  
  
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume
```

*Function comments explain the **purpose** of the function, the meaning of the **parameter** variables and the **return** value, as well as any special **requirements***

## Calling a function

- Once we have defined a function, we can **call** (or **invoke**) it as many times as we like.
- The statements in a function will only be **executed when called**.  
**(store and reuse)**

(1)	x = 5	(1)
(2)	print ('Hello')	(2)
(3)	def print_oks():	(5)
(4)	print ('OK 1')	(6)
(5)	print ('OK 2')	(7)
(6)	print ('World')	(3)
(7)	print_oks()	(4)
(8)	x = x + 2	(8)
(9)	print (x)	(9)

Hello  
World  
OK 1  
OK 2  
7

## Using Functions: Order (1)

- It is important that you define any function before you **call** it
- For example, the following will produce a error:

```
print(cubeVolume(10))
def cubeVolume(sideLength) :
    volume = sideLength ** 3
    return volume
```

- The computer does not know that the cubeVolume function will be defined later in the program

## Using Functions: Order (2)

- However, a function can be called from within another function before the former has been defined.
  - You are not actually **executing** it when called in another function.
- The following is perfectly legal:

```
def main() :  
    result = cubeVolume(2)  
    print("A cube with side length 2 has volume",  
        result)  
  
def cubeVolume(sideLength) :  
    volume = sideLength ** 3  
    return volume  
  
main()
```

## Parameters

- A **parameter** is a variable which we use **in the function definition**.
- It is a “**handle**” that allows the code in the function to **access the arguments** for a particular function call.

### Parameter

```
>>> def greet(lang):
...     if lang == 'es':
...         print ('Hola')
...     elif lang == 'fr':
...         print ('Bonjour')
...     else:
...         print ('Hello')
...
>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('fr')
Bonjour
>>>
```

# Arguments

- An **argument** is a value we pass into the **function** as its **input** when we **call** the function
- Sometimes, the function doesn't need input (**no arguments**)
- We use **arguments** so we can direct the **function** to do different kinds of work when we call it at **different times**
- We put the **arguments** in parentheses after the **name** of the function

```
def calc(a, b) ← Value of a and b
    sum = a + b
    subtraction = a - b
    print ("sum is: ", sum)
    print ("subtraction is: ", subtraction)
```

212

# Arguments

```
x = 3  
y = 5  
m = 2.0  
n = 7  
i = 2.5  
j = 2.2  
calc(x, y) # or calc(3, 5)  
calc(m, n) # or calc(2.0, 7)  
calc(i, j) # or calc(2.5, 2.2)
```

Arguments



Outputs:

```
Sum is: 8  
Subtraction is: -2  
Multiplication is: 15  
Division is: 0  
Sum is: 9.0  
Subtraction is: -5.0  
Multiplication is: 14.0  
Division is: 0.2857142857142857  
Sum is: 4.7  
Subtraction is: 0.3  
Multiplication is: 5.5  
Division is: 1.1363636363636362
```

## Arguments vs. Parameter

- A parameter is a **variable** in a function definition. When a function is called, the arguments are the **actual data** you pass into the function's parameters.

The diagram shows a code snippet in a light gray box. A red arrow labeled "Arguments" points to the call to the function "greet('en')". A green arrow labeled "Parameter" points to the parameter "lang" in the function definition. The code is as follows:

```
>>> def greet(lang):
...     if lang == 'es':
...         print ('Hola')
...     elif lang == 'fr':
...         print ('Bonjour')
...     else:
...         print ('Hello')
...
>>> greet('en')
Hello
>>> greet('es')
Hola
```

## Required arguments

- Required arguments are the arguments passed to a function in correct **order**.
- The number of **arguments** in the **function call** should **match exactly** with the **parameters** (number and order) in the **function definition**.

```
def printme(string):  
    print (string)  
    return  
# call the function  
printme()
```

```
printme("I'm ZZ")
```

```
TypeError: printme() missing 1 required positional  
argument: 'string'
```

```
"I'm ZZ"
```

Required argument

215

# Special cases

- Default argument.
- When you define a function, you may set a default value to your parameter.

## Return Values

- Functions can (optionally) return one value.
  - Add a **return** statement that returns a value
    - A **return** statement does two things:
      - 1) Immediately terminates the function
      - 2) Passes the return value back to the calling function

```
def cubeVolume (sideLength):  
    return 8  
    volume = sideLength * 3  
    return volume ← return statement  
Print(cubeVolume(3))
```

217

## The ***return*** statement

- The ***return*** statement **terminates** the function execution and “**sends back**” the result of the function.
- A function can use multiple return statements.
  - But every branch may have only one return statement.

# Example

```
>>> def greet(lang):
...     if lang == 'es':
...         return 'Hola'
...     elif lang == 'fr':
...         return 'Bonjour'
...     else:
...         return 'Hello'
...
>>> print (greet('en'),'Glenn')
Hello Glenn
>>> print (greet('es'),'Sally')
Hola Sally
>>> print (greet('fr'),'Michael')
Bonjour Michael
>>>
```

What greet does:

- 1) Receives the argument value
- 2) If the lang is 'es', then return 'Hola', otherwise,
  - 2.1) If the lang is 'fr', then return 'Bonjour', otherwise return 'Hello'

## Make Sure A Return Catches All Cases

- Missing return statement
  - Make sure all conditions are handled
  - In the `cubeVolume` case, `sideLength` could be 0
    - No return statement for this condition
    - The computer will *not* complain if any branch has no return statement
    - It may result in a run-time error because Python returns the special value `None` when you forget to return a value

```
def cubeVolume(sideLength) :  
    if sideLength > 0 :  
        return sideLength ** 3  
print(cubeVolume(2))  
print(cubeVolume(0))
```

220

## Make Sure A Return Catches All Cases

- A correct implementation:

```
def cubeVolume(sideLength) :  
    if sideLength > 0  
        return sideLength ** 3  
    else :  
        return 0
```

221

## Alternative return example

```
def cubeVolume(sideLength) :  
    if sideLength > 0:  
        volume = sideLength ** 3  
    else :  
        volume = 0  
    return volume
```

- Alternative to multiple returns (e.g., one for each branch):
  - You can avoid multiple returns by storing the function result in a variable that you return in the last statement of the function.

222

## Functions Without **return** Values

- Functions are not required to return a value
  - No return statement is required
  - The function can generate output even when it doesn't have a return value

```
...  
boxString("Hello")  
...  
-----  
!Hello!  
-----
```

```
def boxString(contents) :  
    n = len(contents) :  
    print("-" * (n + 2))  
    print("!" + contents + "!")  
    print("-" * (n + 2))
```

## Using `return` Without a Value

- You can use the `return` statement without a value
  - The function will terminate immediately!

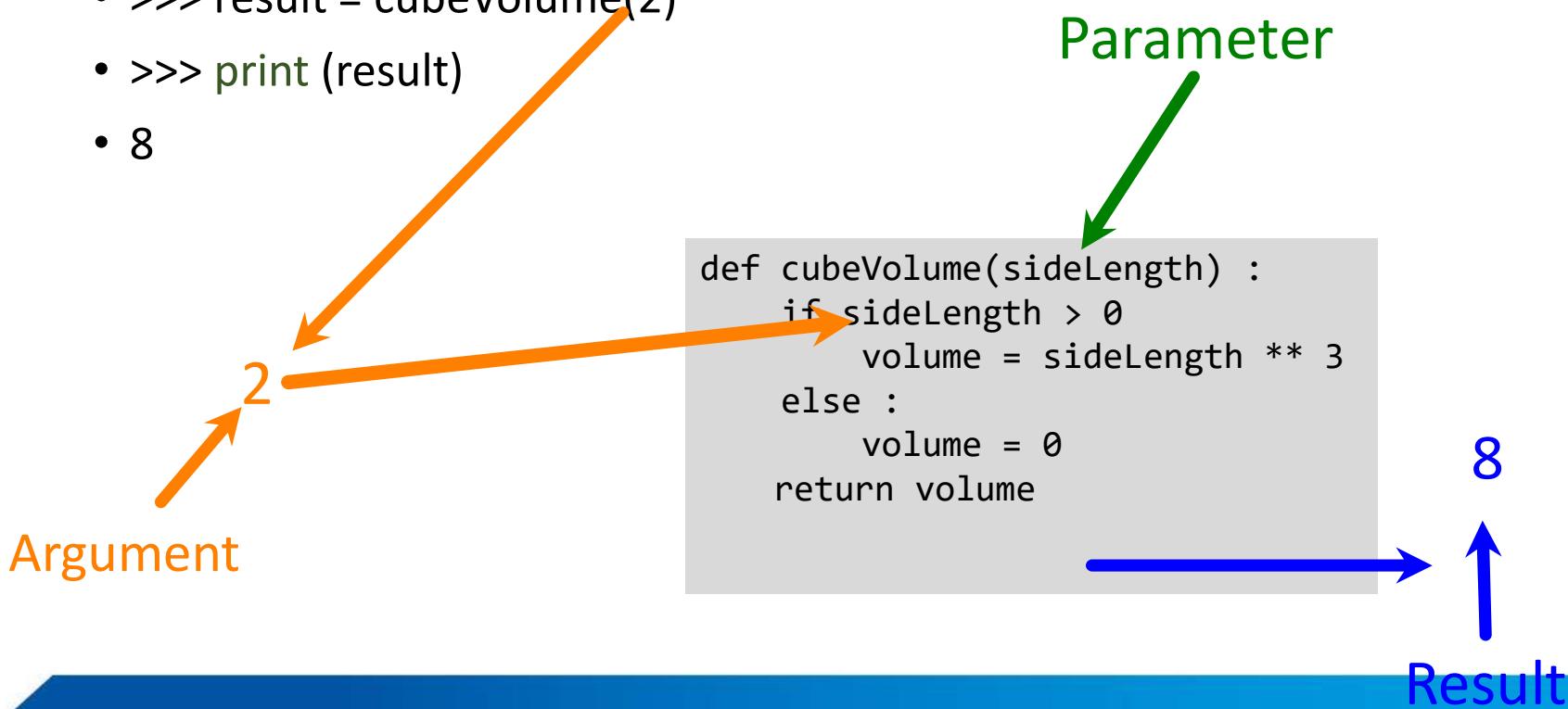
```
def boxString(contents) :  
    n = len(contents)  
    if n == 0 :  
        return # Return immediately  
    print("-" * (n + 2))  
    print("!" + contents + "!")  
    print("-" * (n + 2))
```

## With or without return?

- Return if you need the function to give you a result to be used later.
- No return if you need the function to do something for you.
  - It may be a good idea to return an [temporal] indicator.

## Arguments, parameters, and results

- >>> result = cubeVolume(2)
- >>> print(result)
- 8



## Multiple parameters / arguments

- We can define more than one **parameter** in the **function** definition
- We simply add more **arguments** when we call the **function**
- We match the number and the order of arguments and parameters

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print (x)

y = addtwo(3)
print(y)
```

# Exercise

- Write a program to find out all the leap years in a given time period (for example, 2001 to 2100), and store your results in a list. When doing so, design a function to identify all the leap years in a time period.

## Recursion

- It is legal for one function to call another.

```
def distance(x1, y1, x2, y2):
    dx = subtraction(x2,x1)
    dy = subtraction(y2,y1)
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result

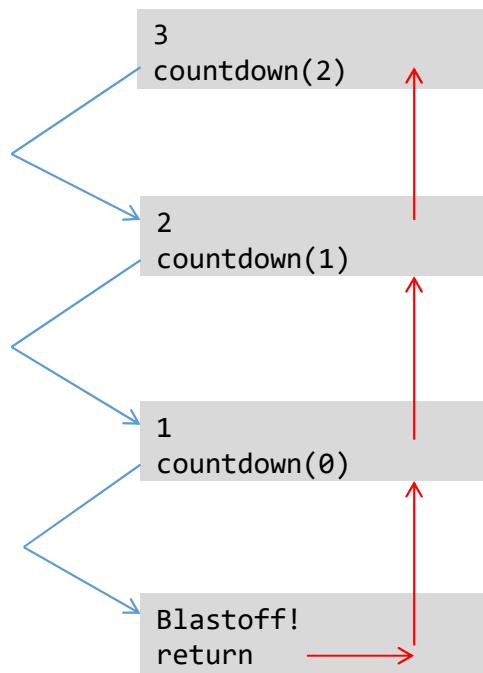
def subtraction(a, b):
    sub = a-b
    return sub
```

- What about calling itself?

## Recursion

```
def countdown(n):
    if n == 0:
        print("Blastoff!")
    else:
        print (n)
        countdown(n-1)

>>> countdown(3)
```



## Recursive Functions

- A recursive function is a function that calls **itself**
- A recursive computation solves a problem by using the solution of the same problem with updated (“simpler”) inputs
- For a recursion to terminate, there must be special cases for the updated (“simpler”) inputs

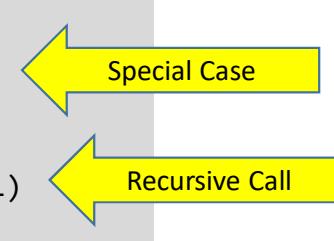
## Recursive Triangle Example

- The function will call itself (and not output anything) until sideLength becomes < 1
- It will then use the return statement and each of the previous iterations will print their results
  - 1, 2, 3 then 4

```
def printTriangle(sideLength) :  
    if sideLength < 1 :  
        return  
    printTriangle(sideLength - 1)  
    print("[]" * sideLength)
```

```
[]  
[] []  
[] [] []  
[] [] [] []
```

Print the triangle with side length 3.  
Print a line with four [].



```
def printTriangle(sideLength) :  
    if sideLength < 1 :  
        return  
    printTriangle(sideLength - 1)  
    print("[]" * sideLength)  
  
printTriangle(4)  
printTriangle(3)  
    printTriangle(2)  
        printTriangle(1)  
            printTriangle(0)  
                return  
                print("[]" * 1)  
            print("[]" * 2)  
        print("[]" * 3)  
print("[]" * 4)
```

Output:  
[]  
[][]  
[][][]  
[][][][]

## Recursive Calls and Returns

Here is what happens when we print a triangle with side length 4.

- The call `printTriangle(4)` calls `printTriangle(3)`.
  - The call `printTriangle(3)` calls `printTriangle(2)`.
    - The call `printTriangle(2)` calls `printTriangle(1)`.
    - The call `printTriangle(1)` calls `printTriangle(0)`.
      - The call `printTriangle(0)` returns, doing nothing.
      - The call `printTriangle(1)` prints `[]`.
    - The call `printTriangle(2)` prints `[][]`.
    - The call `printTriangle(3)` prints `[][][]`.
  - The call `printTriangle(4)` prints `[][][][]`.

```
def printTriangle(sideLength) :  
    if sideLength < 1 :  
        return  
    printTriangle(sideLength - 1)  
    print("[" * sideLength)
```

```
def countdown(n):  
    if n == 0:  
        return "Blastoff!"  
    else:  
        print (n)  
        countdown(n-1)
```

# Exercise

- Write a program that returns the sum of the first n integers. n is user's input.

# **File Operation**

## Text file processing

- A text file can be thought of as a sequence of lines
  - A text file (sometimes spelled "textfile": an old alternative name is "flatfile") is a kind of computer file that is structured as **a sequence of lines** of electronic text.
  - Text file may not necessarily just be .txt file. It can be others like .py file, .html file or even .csv file.
  - The other type of file is binary file. Binary file can only be processed by special program.

## Open a file

- Text file can be accessed by Python directly using build-in function.
- Before we can read the contents of the file we must tell Python which file we are going to work with and what we will be doing with the file
- This is done with the `open()` function
- `open()` returns a “`file handle`” – a file object used to perform operations on the file
- Kind of like “File -> Open” in a Word Processor

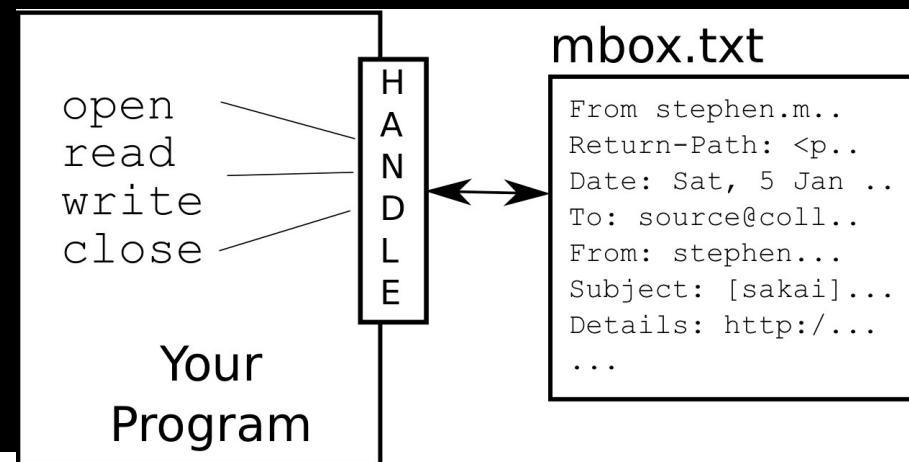
239

# Open()

- Syntax
  - ❑ `file_handler_variable = open(filename, [mode])`
  - ❑ returns a **file object** used to manipulate the file
  - ❑ `filename` is a **string** (a variable with string value)
  - ❑ `mode` is optional and should be '**r**' if we are planning reading the file and '**w**' if we are going to write to the file.

## What is a handler?

```
>>> fh = open('mbox.txt','r')
>>> print (fh)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp936'>
```



241

## File object as a sequence

- A **file object** open for read can be treated as a **sequence** of strings where each line in the file is a string in the sequence
- We can use the **for** statement to **iterate** through a **sequence**
- Remember - a **sequence** is an ordered set

```
file_wbs = open('wbs.txt', 'r')
for line in file_wbs:
    print (line)
```

## Read the ‘whole’ file

- We can **read** the whole file (newlines and all) into a **single string**.

```
file_wbs = open('wbs.txt', 'r')
file_lines = file_wbs.read()
print(type(file_lines))
print (len(file_lines))
print (file_lines[:20])
```

## Reading Multiple Lines From a File

- We can use `readlines()` to get a list of lines.
- Each element in the list is a line.

```
file_wbs = open('wbs.txt')
lines = file_wbs .readlines()
print(type(lines))
print (len(lines))
print (lines[:2])
['the first line', 'the second line']
```

## Files Tips: Reading

- Important things to keep in mind:
  - When opening a file for reading, the file must **exist** (and otherwise be **accessible**) or an exception occurs
  - The file object returned by the open function must be **saved in a variable** (assign the file object to a variable)
    - All operations for accessing a file are made via the file object

```
file_wbs = open('wbs.txt', 'r')
file_lines = file_wbs.readlines()
file_all = file_wbs.read()
```

## Write to file

- The write() method writes any **string** to an open file.
- The write() method does not add a newline character ('\n') to the end of the string

```
>>> fh = open('test.txt', 'w')
>>> lines = fh.readlines()
>>> fh.write('Python is great\nI like Python')
>>> fh.close()
```

246

## Closing File: Important

- When you are done processing a file, be sure to *close* the file using the `close()` method:

```
fh.close()
```

- If your program exits without closing a file that was opened for writing, some of the output may not be written to the disk file.

## Now, try this

```
fh = open('test.txt', 'w')
fh.write('I also like Java')
fh.close()
```

## Files Tips: Writing

- If the output file already exists, it is **emptied** before the new data is written into it with “**w**” mode.
- If the file does not exist, an empty file is created.

## Append to a file

- Another mode to open a file, other than ‘r’ and ‘w’, is ‘a’, which means open to append at the end of the file.
- Similar to ‘w’, string can be written to the file. However, instead of emptying existing content, it will append to existing content.

```
file_wbs = open('wbs.txt', 'a')
file_wbs.write('1234')
```

## Different mode in open()

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	open for exclusive creation, failing if the file already exists
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)

# Exercise

- You have a text file with two lines, one for the start-year and another one for the end-year. Find all the leap years between the two years and write them into another text file.

# Exception Handling

# Exception Handling

- There are two aspects to dealing with run-time program errors:
  - 1) Detecting Errors
  - 2) Handling Errors
- The open function can detect an attempt to read from a non-existent file
  - The open function cannot handle the error
  - There are multiple alternatives, the function does not know which is the correct choice
  - The function reports the error to another part of the program to be handled
- Exception handling provides a flexible mechanism for passing control from the point of the error to a **handler** that can deal with the error

# Detecting Errors

- What do you do if someone tries to withdraw too much money from a bank account?
- You can **raise** an exception
- When you **raise** an exception, execution does not continue with the next statement
  - It transfers to the **exception handler**

Use the `raise` statement  
to signal an exception

```
if amount > balance :  
    raise ValueError("Amount exceeds balance")
```

# Exercise

d = 10/0

a = 5

b = 5

c = 10

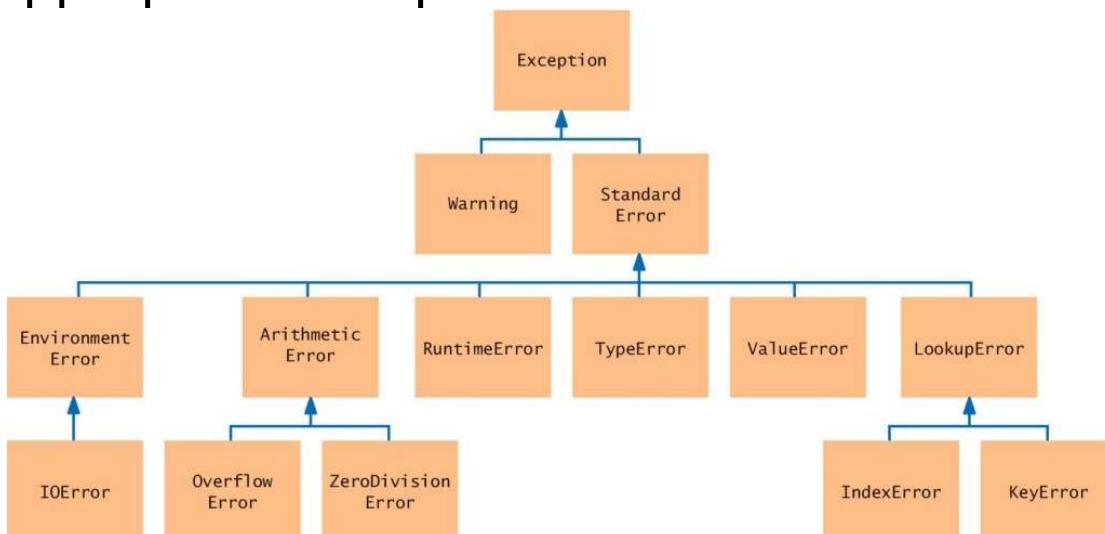
if a-b == 0:

```
    raise ZeroDivisionError("b cannot equal to a")
```

d = c / (a-b)

# Exception Classes (a subset)

Look for an appropriate exception



<https://docs.python.org/3/library/exceptions.html#Exception>

# Syntax: Raising an Exception

**raise** exceptionObject[*(message)*]

```
a = 5
b = 6
if a<b:
    raise ValueError("wrong number")
    print("a is smaller than b")
```

# Tips for Handling Exceptions

- Every exception should be handled somewhere in the program
- This is a complex problem
  - You need to handle each possible exception and react to it appropriately
- Handling recoverable errors can be done:
  - Simply: exit the program
  - User-friendly: Ask the user to correct the error

# Handling Exceptions: Try-Except

- You handle exceptions with the `try/except` statement
- Place the statement into a location of your program that knows how to handle a particular exception
- The `try` block contains one or more statements that `may` cause an exception of the kind that you are willing to handle
- Each `except` clause contains the handler for an exception type

# Syntax: Try-Except

```
try:  
    statement 1  
    statement 2  
    ...  
except ExceptionType:  
    statement 1  
    ...  
except:  
    statement 1  
    ...  
raise
```

# Try-Except: An Example

```
try :  
    filename = input("Enter filename: ")  
    infile = open(filename, "r")      open() can raise an IOError  
    line = infile.readline()  
    value = int(line)               int() can raise a  
    . . .  
    . . .  
except IOError :  
    print("Error: file not found.")  
except ValueError as exception :  
    print("Error:", str(exception))  
    Execution transfers here if  
    file cannot be opened  
    Execution transfers here  
    if the string cannot be  
    converted to an int
```

*If either of these exceptions is raised,  
the rest of the instructions in the try  
block are skipped*

# Example

- If an IOError exception is raised, the **except** clause for the IOError exception is executed
- If a ValueError exception occurs, then second **except** clause is executed
- If any other exception is raised it will not be handled by any of the **except** blocks

# Example

```
a = 5
```

```
b = 5
```

```
c = 10
```

```
if a-b == 0:
```

```
    raise ZeroDivisionError("b cannot be the same as a")
```

```
d = c / (a-b)
```

# Example

a = 5

b = 5

c = 10

try:

    d = c / (a-b)

except ZeroDivisionError:

    print("b cannot be the same as a")

    a = input()

# **Modules**

266

# Exercise

Take user input for start and end year. If the inputs are not numbers, inform the user to re-enter the correct ones. Then find all the leap years between the start and end year and write the leap years into a text file.

## Modules

- collection of definitions and statements (functions and variables) that can be imported.
- Many build-in modules: `math`, `random`, `os`, etc.
- collection of blackboxes
- Package: collection of modules

## Create own module

- You can also create your own module, similar to the way you create your own Python script.
- Write your functions and save as .py file.
- The file name will be the module name.

- A module called **fibo** saved as fibo.py with two functions: func1() and func2().

```
# Fibonacci numbers module
def func1(n):  # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()
def func2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

270

## Use module (1)

- Import the module as a whole (only module name is imported into local name space):

```
import module_name
```

```
import fibo
```

- Allow you to access the functions in the module:

```
module_name.function_name
```

```
fibo.func1(1000)
```

- You can also give a local name:

```
F_local = fibo.func1
```

```
F_local(500)
```

## Use module (2)

- Directly import certain functions in the module (only function names are imported into local name space).

```
from module_name import function1, function 2...
```

```
from fibo import func1
```

```
func1(1000)
```

- You can even import all functions in the module.

```
from module_name import *
```

```
from fibo import *
```

```
fiunc1(1000)
```

## Module search path

- current directory
- list of directories specified in PYTHONPATH environment variable
- uses installation-default if not defined, e.g.,  
  /usr/local/lib/python  
  /Anaconda/lib/site-packages
- uses sys.path
  - >>> import sys
  - >>> sys.path

273

# Python virtual environment

- Modules are easily broken and/or conflicting.
  - Be caution when updating your packages
  - The dependencies will be updated.
- Use Python virtual environment to manage the development.
  - a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages.
  - Each environment is independent from others.
  - Easy to manage packages for specific application.

# Environment in Anaconda

- Create environment using Anaconda-Navigator.
  - A root environment is created by default.
  - Install Spyder/Jupyter for newly created environment.
- To check existing environment.

`conda env list`

```
(base) C:\Users\zzw>conda env list
# conda environments:
#
base          * D:\Anaconda
bigquery      D:\Anaconda\envs\bigquery
keras         D:\Anaconda\envs\keras
nlp           D:\Anaconda\envs\nlp
web           D:\Anaconda\envs\web
```

- <https://conda.io/docs/user-guide/tasks/manage-environments.html>

## Install new modules

- Generally, Anaconda will handle the installation of new modules.

`conda list`

- In prompt or terminal, type

`conda install module_name`

`conda install scipy`

- <https://conda.io/docs/user-guide/tasks/manage-pkgs.html>

(source) activate *root(or other environment name)*

`pip install scipy`

276

## Example: CSV file operation

- A comma-separated values (CSV) file stores **tabular data (numbers and text) in plain text**.
- Similar to previous text file, CSV files are also structured as a sequence of **formatted lines**.
- Each line of the file is a data record. Each record consists of one or more fields, separated by commas.
- CSV file can be opened by many software, including Excel, but it differs from Excel file.

## Use csv module

- In order to operate csv file, you need to import build-in module “csv” before call the functions in that module to manipulate csv file.

*import csv*

- To read the data, use csv.reader()
- To write the data, use csv.writer()

## Read the data in csv file

- `csv.reader(file_handler, [delimiter=','], [quotechar='|']...)`
- Return a **reader object** which will iterate over lines in the given csv file. (a collection of ordered data)

```
file_wbs = open('wbs.csv')
spamreader = csv.reader(file_wbs)
print(spamreader)
for row in spamreader:
    print(row)
    print(row[0])
```

279

## Write the data to csv file

- `csv.writer(file_handler, [delimiter=','], [quotechar='|']...)`
- Return a **writer object** responsible for converting the user's data into delimited strings
- **writerow()** is a method of writer to insert one row at the end of the csv file. Row should be a **list** of data

```
file_wbs = open('wbs.csv','a')
spamwriter = csv.writer(file_wbs)
for i in range(3):
    spamwriter.writerow([i, '123', '456'])
```

280

## Newline

- In some operation system (like Windows), csv.writer() may add extra line in between rows. To solve that, when open the file to write, add an optional parameter *newline=""*

```
file_wbs = open('wbs.csv','a',newline="")
spamwriter = csv.writer(file_wbs)
for i in range(3):
    spamwriter.writerow([i, '123', '456'])
```

# Exercise

- Write a program to check every year in a given time period (for example, 2001 to 2100) if it is a leap year, and store your results in a csv file (leap.csv). When doing so, design a function to check the years.
- Example

Input start: 2000

Input end: 2010

In your csv file, your data should look like:

2000,yes

2001,no

2002,no

# **Review: basic**

- What is a variable?
- Why do we use variable?
- How to name a variable?

# Review: statement flow

- Sequential flow
- Conditional flow
- Loop flow

# Conditional flow

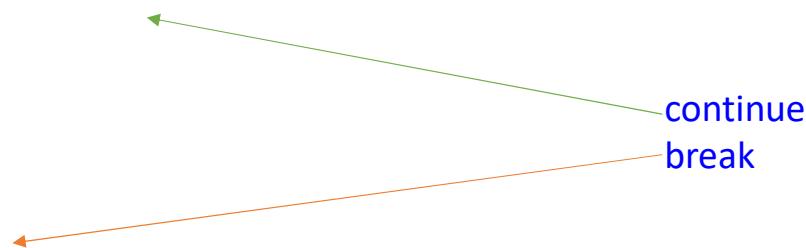
- Format
- `if` condition:
  - statement body
- `elif` condition:
  - *statement body*
- ...
- `else`:
  - *statement body*

x	and	y	Returns
True	and	True	
True	and	False	
False	and	True	
False	and	False	

x	or	y	Returns
True	or	True	
True	or	False	
False	or	True	
False	or	False	

# Loop flow: while

- while condition:
  - statement body
  - statements



## Loop flow: for

- `for iterator in iterable_object:`
- statement body

# List, tuple and string

- They are objects as a collection of sequence data, which is iterable.
- List is **mutable** while tuple and string are **immutable**.
- Indexing and slicing

# Function

- A blackbox that contains a block of organized, reusable code.
- Better modularity and a high degree of code reusing.
- Parameter, argument and return value.

- `def function_name([parameters]):`
- statement body
- [return]
- Function needs to be **defined** before **called**

# File operation

- For text file
  - 1. Create a file object using `open("file_name","mode")`
  - 2. read or write to the file
    - 1. `read()`
    - 2. `readlines()`
    - 3. `write()`
  - 3. Close the file after operation by `close()`

- For csv file
  - 1. Import csv module
  - 2. Create a file object using open("file\_name","mode")
  - 3. read or write to the file
    - 1. Create a reader or writer object
    - 2. Iterative to access the object
  - 4. Close the file after operation by close()

# Module

- Install module: pip vs. conda
- Import a module or functions:
- `import module_name`
- `from module_name import function_name`

**zhewei.zhang@wbs.ac.uk**

# Data Collection

# Collecting publicly available data

- Increasing number of digital trace data that are publicly available online.
- Two major approaches to collect those data:
  1. Web scraping
  2. APIs
  3. Open data

# Some basis of HTML

- Webpages are created using **HTML (Hypertext Markup Language)**, with CSS (Cascading Style Sheets) and JavaScript.
- HTML is a standard markup language (others like XML, LaTex) that uses **tags** to annotate the webpage content so they can be displayed as desired.

# A simple html example

```
<!DOCTYPE html>
<html>
<head>
<title>A simple html example</title>
</head>
<body>
<p id="first1"> Zhewei </p>
<p id="last1"> Zhang </p>
</body>
</html>
```

# A simple html example

```
<!DOCTYPE html>
<html>
<head>
<title>A simple html example</title>
</head>
<body>
<p class = "name" id="first1"> Zhewei </p>
<p class = "name" id="last1"> Zhang </p>
</body>
</html>
```

- Most **tags** will be used in pairs with opening tag and closing tag, i.e. `<head>` and `</head>`.
- Content between a pair of opening and closing tags will be displayed accordingly, which is also called **element** in a html file.

# A simple html example

```
<!DOCTYPE html>
<html>
<head>
<title>A simple html example</title>
</head>
<body>
<p class = "name" id="first1"> Zhewei </p>
<p class = "name" id="last1"> Zhang </p>
</body>
</html>
```

- All HTML elements can have **attributes** that provide additional information about that element.
- Attributes are always specified in the **opening tag**.
- For web scraping, we normally deal with two kinds of attributes: **id** and **class**.
  - **id** can uniquely identify individual elements.
  - **class** can identify a set (class) of similar elements.

# Web scraping in two steps

1. Access the webpage and download the html file.



- **Requests**
- <https://2.python-requests.org/en/master/user/quickstart/>

2. Extract elements (data) from a webpage, which can be located by the tags and attributes.



- **BeautifulSoup**
- <https://www.crummy.com/software/BeautifulSoup/>

# Fetch the webpage with Requests

- Requests is a HTTP library for Python that help you to make **http request** to the web server and fetch the webpage.
- You send a request to the web server asking for certain webpages.
- You can make **get** and **post** request.

```
import requests  
url = "test.html"  
result = requests.get(url)
```

# Requests results

- The `get()` function will send a http request to retrieve the webpage specified. The only required argument for `get()` is the `url` address to the webpage.
- Returns a requests object that contains various information about the webpage retrieved with attributes:
  - `status_code`: 200 is good and 4xx (403,404,408,etc.) is bad.
  - `encoding`: how characters are coded digitally, utf-8 is generally the best.
  - `text`: the plain content of the webpage, including all markings.

# Retrieve links with parameters

- Some webpages, such as search results, may require you to send necessary **parameters**.
- `http://yourmainlink.com?name=python&format=ebook&lang=en`
- You can pass those parameters by using a **dictionary** as the argument **param**.
- `payload = {'key1' : 'value1', 'key2' : 'value2'}`
- `payload = {'name' : 'python', 'format' : 'ebook', 'lang' : 'en'}`
- `result = requests.get(url, params=payload)`

## Deal with limits

- Regardless legal issue, websites generally do not like scraping as it can take up the bandwidth. Therefore, many websites limit the number of requests you can make within certain time period. Your requests can be blocked after reaching the limit.
- So, be **considerate**, and **pause** a while if needed.
- proxy and headers can be provided as additional arguments to the `get()`.

# Parsing your result with BeautifulSoup

- You need to **parse** the text retrieved by Requests: convert the plain text into structured data.
- BeautifulSoup is a parser library to create such **tree-structured** data by parsing HTML or XML files.

```
from bs4 import BeautifulSoup  
url_html = result.text  
url_content = BeautifulSoup(url_html, 'html.parser')
```

# Navigate the parsed data

- Parsed result will be return as a BeautifulSoup object that has a range of methods to help you navigate and search the data.

1. Using tag names directly.

```
url_content.head # return the first matching element.
```

2. Using `find()` and `find_all()`

```
url_content.find_all('p') # find_all() returns a list
```

3. Using `select()`

```
url_content.select() # use CSS selectors instead of HTML tags.
```

# Exercise

- Write a script to extract FTSE indices from London Stock Exchange.
- <https://www.londonstockexchange.com/exchange/prices-and-markets/stocks/indices/ftse-indices.html>

# Other thoughts

- Check **robots.txt** before scraping the data. It is basically an instruction telling web robot Dos and Don'ts. This can be found by adding it to the url of the main site. For example, <https://www.warwick.ac.uk/robots.txt>.
- User agent spoofing. Sometimes, the web server will not response if no user agent is provided. You may manually create one or use some library (e.g. fake-useragent) to generate a random one and pass to the **header** argument.
- Set a timeout to avoid indefinitely waiting by passing a value (in seconds) to argument **timeout**.

# Downloading data through APIs

- Many company provide APIs for developer to access their data.
- Many third parties also develop API-like libraries for data collection.
- Some considerations include:
  - Accuracy.
  - Ease of use.
  - Eligibility.
  - Limitation.
  - Cost.

# Open datasets

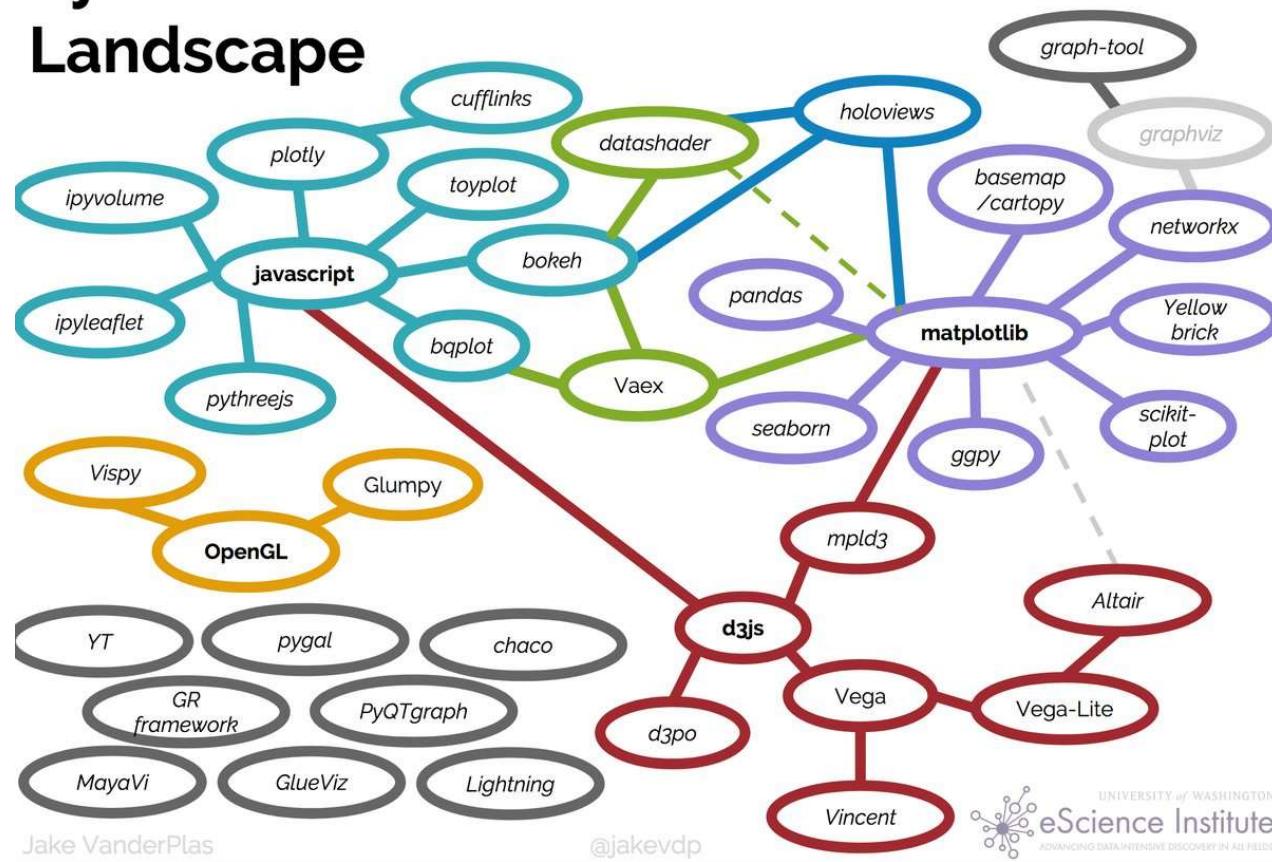
- There are many open datasets that you can use together with your internal dataset and collected dataset.
- <https://toolbox.google.com/datasetsearch>
- <https://trends.google.com/trends/explore>
- <http://data.europa.eu/euodp/en/data/>
- <https://opencorporates.com/>

# Data Visualization in Python

# Data visualization

- Eyeballing is a good way to have some quick initial understanding of your data. It is often used as the first step to start working on the data you just collected.
- There are tons of data visualization libraries for Python. Many data analysis libraries also provide built-in visualization functions.

# Python's Visualization Landscape



# Matplotlib and Seaborn

- **Matplotlib** is a plotting library for Python. It is the basis of many other libraries' visualization functionality. General and powerful yet a bit tricky sometimes.
- **Seaborn** is a data visualization library based on Matplotlib. It focuses on statistical visualization and provides easier, more user-friendly way to generate prettier graphs.
- We use build-in datasets for demonstration. You can explore these datasets here: <https://github.com/mwaskom/seaborn-data>

# Tips dataset

total_bill	tip	sex	smoker	day	time	size
16.99	1.01	Female	No	Sun	Dinner	2
10.34	1.66	Male	No	Sun	Dinner	3
21.01	3.5	Male	No	Sun	Dinner	3

```
import seaborn as sns  
tips = sns.load_dataset("tips")
```

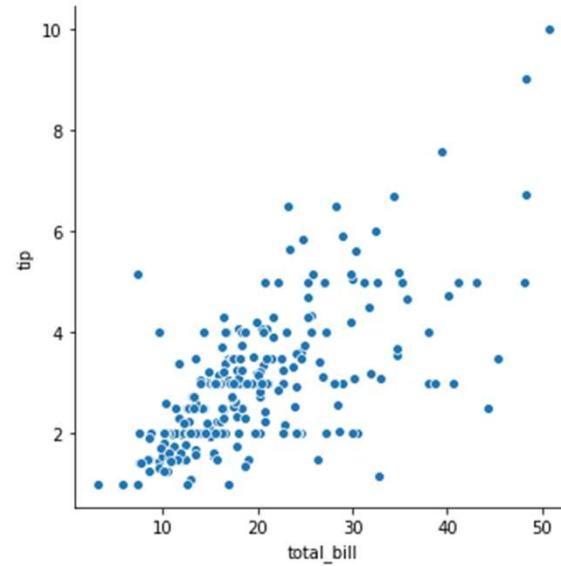
*Function names*  
**Important things**  
**Parameter names**  
**Values for argument**

# Scatter Plots

- Scatter plot is one of the most used visualization to explore the relationships between (two) variables.
- `relplot()` is a seaborn function that draws relational plots.

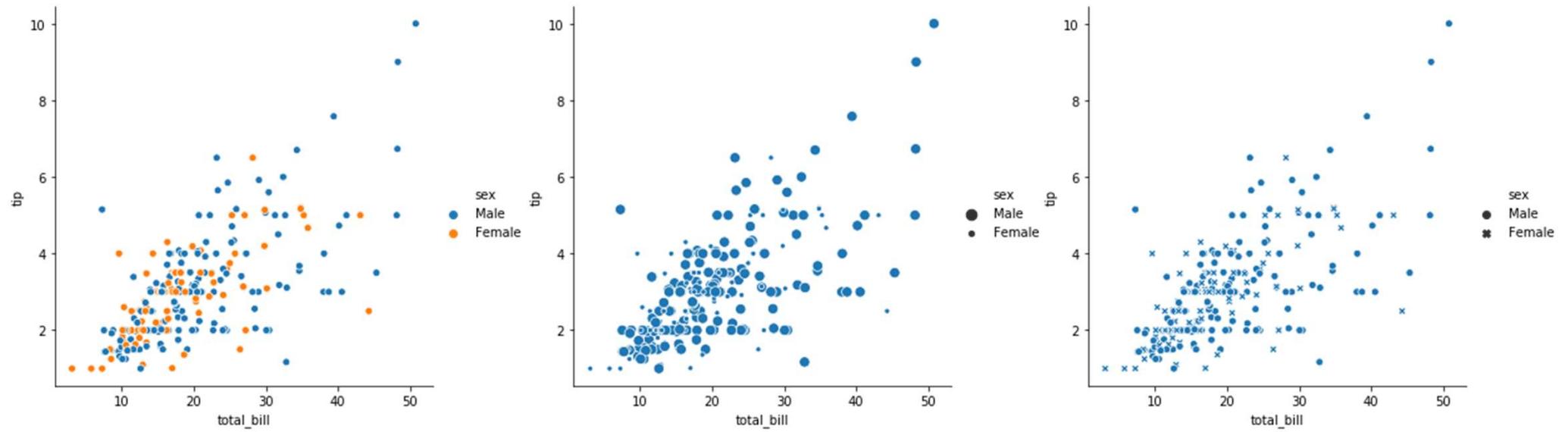
```
sns.relplot(x="total_bill",
y="tip", data=tips)
```

#three required arguments: x variable name, y variable name and the dataset name. *Values of x and y need to be numeric.*



# Optional arguments

- You may specify additional variables from the dataset to be used as the extra dimensions to group your data points, by passing the variable name to the arguments:
  1. `hue`: your data will be group in different `colours`.
  2. `size`: your data will be group in different `size`.
  3. `style`: your data will be group in different `style`.
  4. They can be used together.



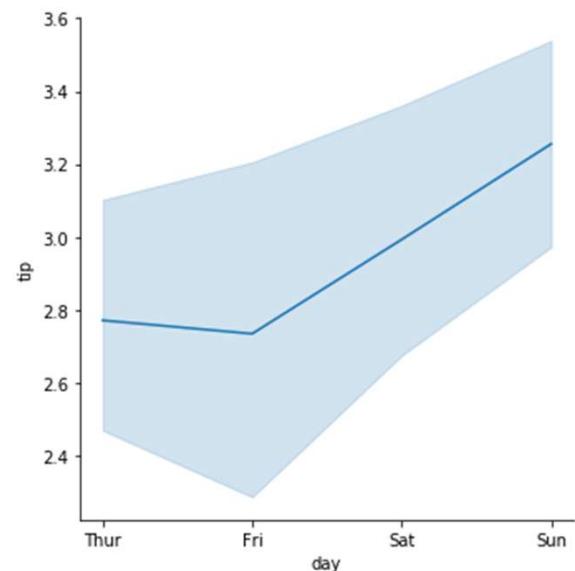
# Exercise

- Draw a scatter plot about the relationship between tip and total bill. Explore how different factors, such as smoker, day, time, size may affect this relationship.

# Line plot

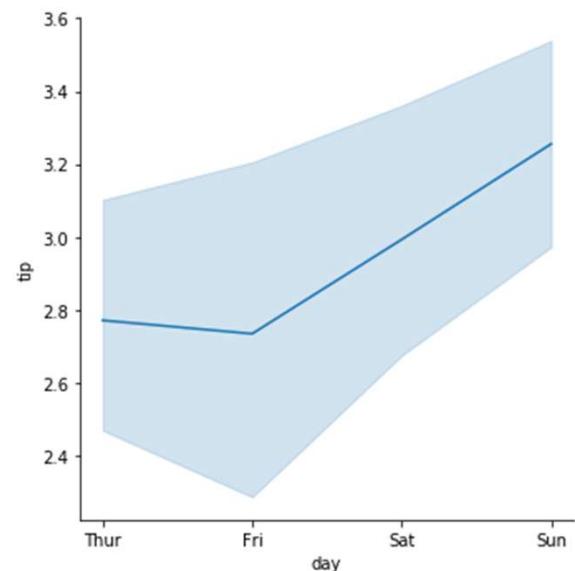
- When you have continuous variables, such as time, it can be a good idea to view the changes with line plot.
- An optional argument in `relplot()` is `kind`, with default value "scatter" for scatter plot, can be passed with value "line" to draw a line plot.

```
sns.relplot(x="day", y="tip",  
kind='line', data=tips)
```

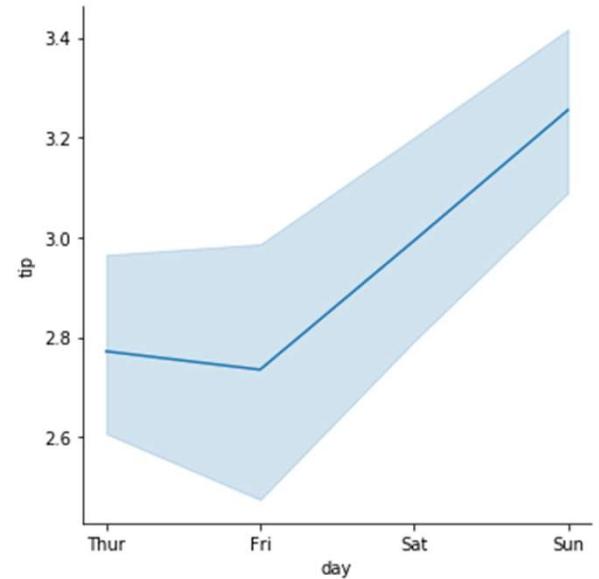
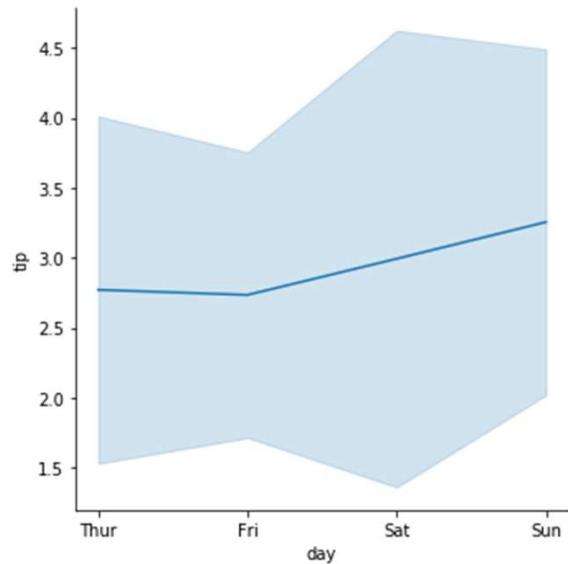
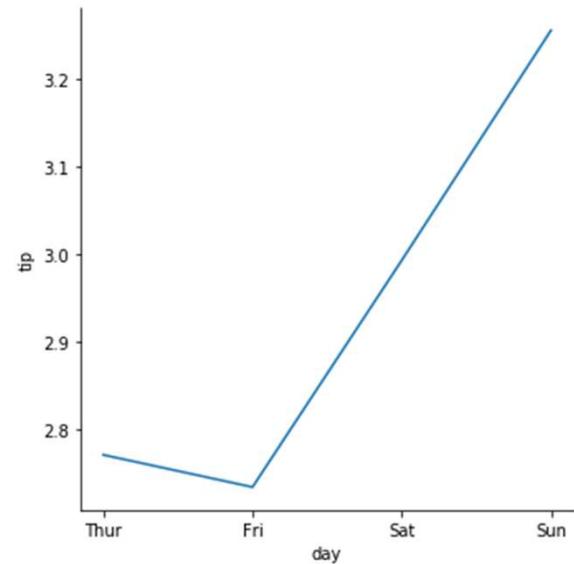


# Line plot

- By default, Seaborn aggregates the multiple measurements at each x value by plotting the **mean** and the 95% **confidence interval** around the mean
- You can change both by passing arguments:
  1. `ci`: value, **none**, or '`sd`'.
  2. `estimator`: **none** or other pandas estimator.

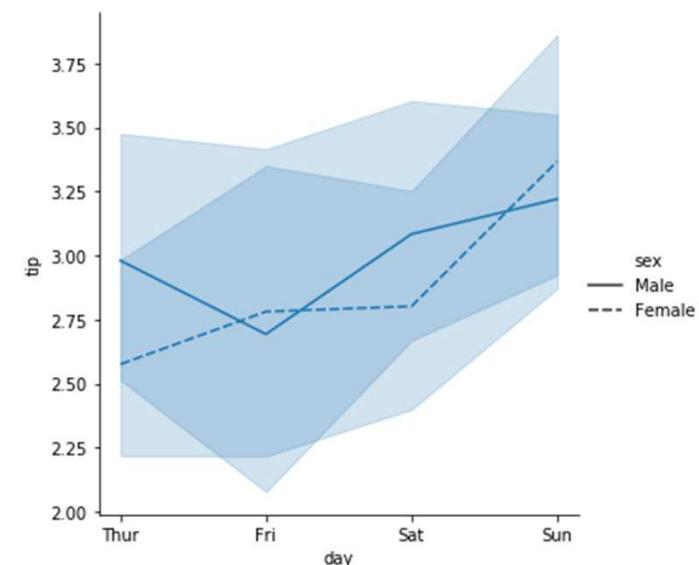
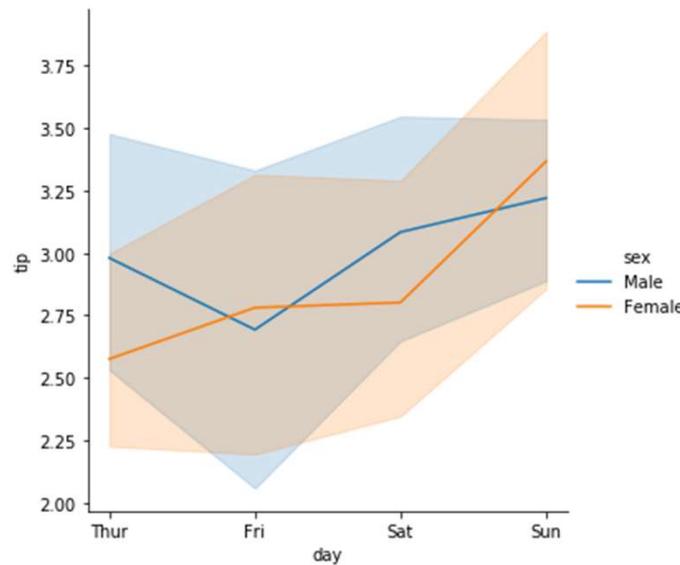
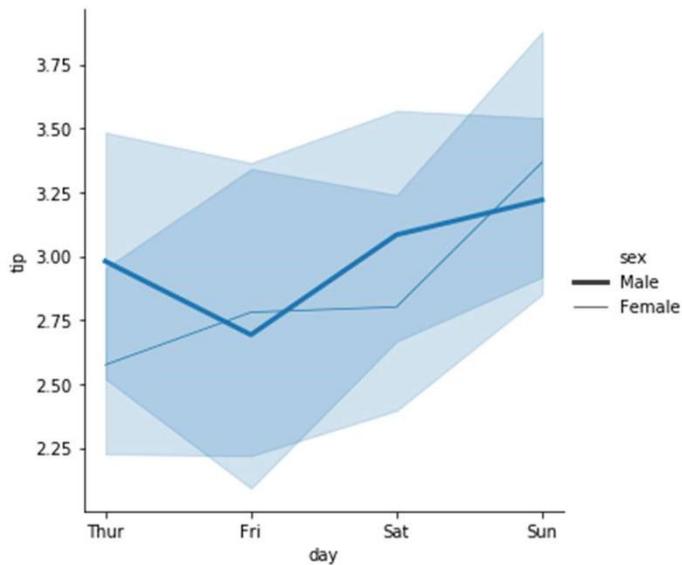


# Different confidence intervals



# Line plotting by groups

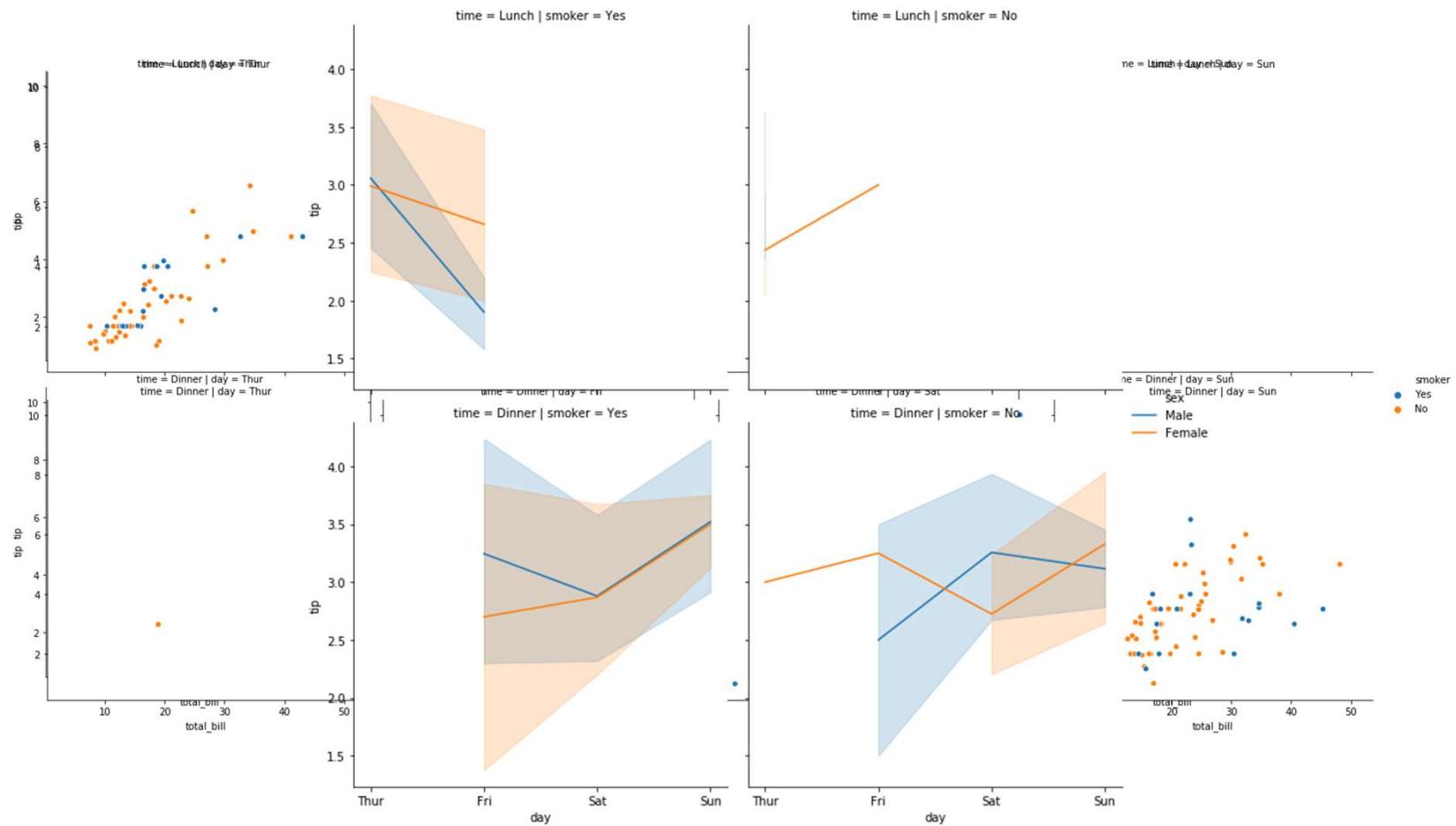
- You can also create multiple lines based on the groups spitted by hue, size and style.



# Showing multiple relationships

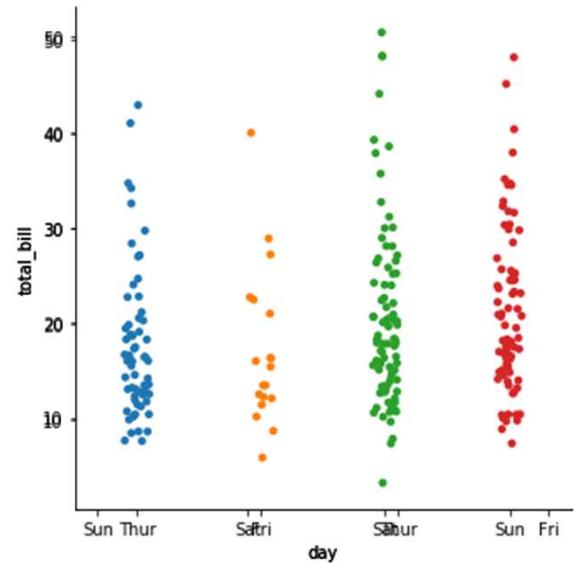
- Very often, when you explore your dataset, you would like to compare multiple relationships at once.
- You can create a grid of graphs and specify facets (grouping dimensions) by passing arguments:
  1. `col`: the variable used for splitting in horizontal direction.
  2. `row`: the variable used for splitting in vertical direction.

```
sns.relplot(x="total_bill", y="tip", col='day',
row='time', data=tips)
```

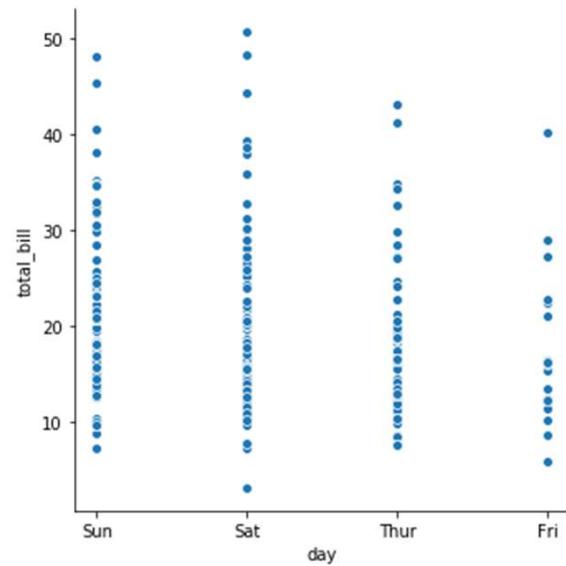
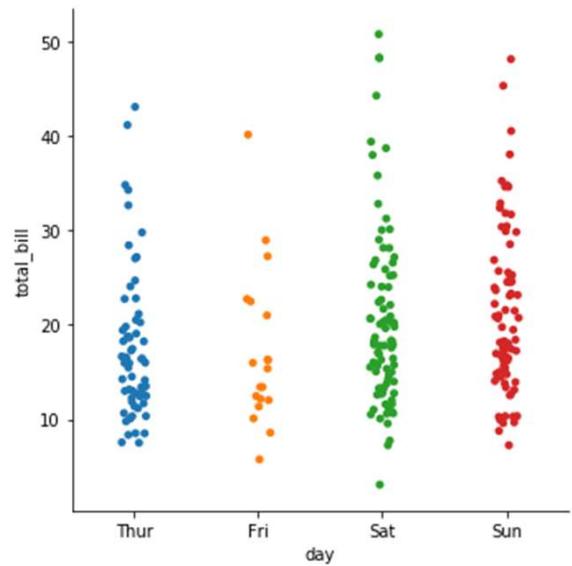


# Plotting categorical variables

- `relplot()`, while best to plot relationship between numeric variables, is able to handle categorical variables.
- `catplot()` is a function dedicated for categorical plotting.
- `sns.catplot(x="day", y="total_bill", data=tips)`

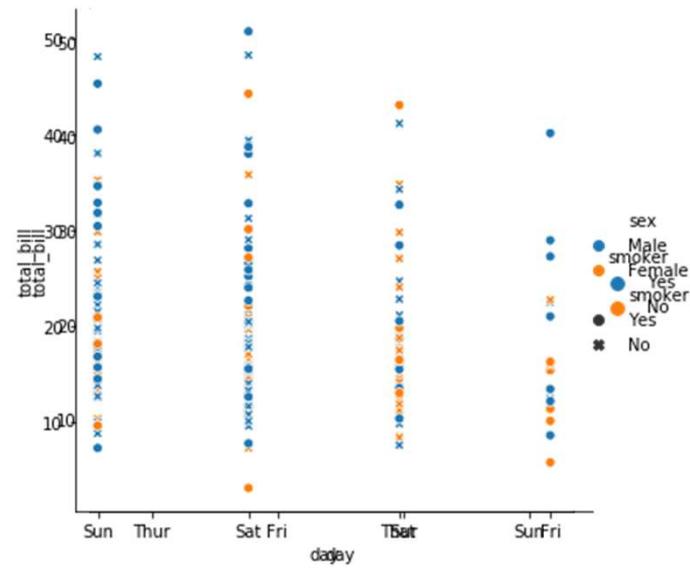


# Spot three differences?



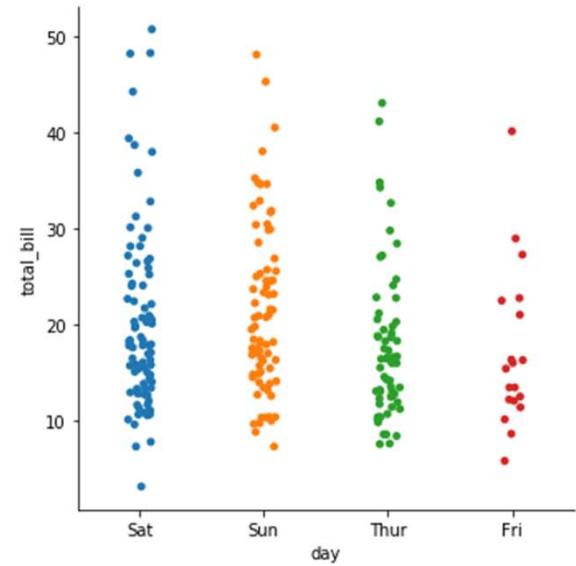
# Adding dimension

- *catplot()* only support hue to add another dimension.
- If you need add more dimensions with different colours, styles or sizes, you can use *relplot()* instead.



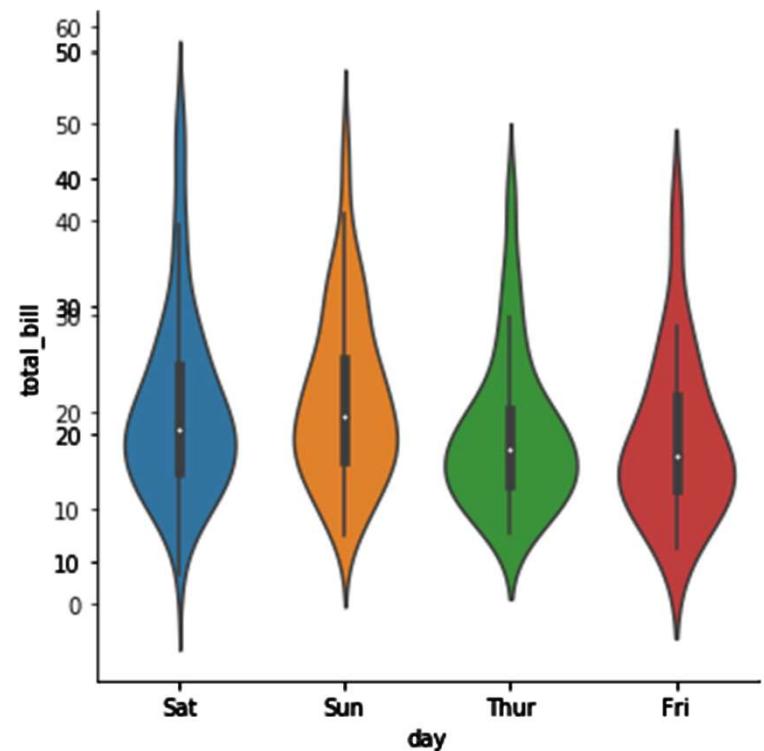
# Order the category

- By default, seaborn will try to infer the order of categories from your data, such as numerical order.
- However, it does not always work well. So we can manually set the order by passing a list of values to argument: `order`
- `sns.catplot(x="day",  
y="total_bill", order=['Sat',  
'Sun', 'Thur', 'Fri'],  
data=tips)`



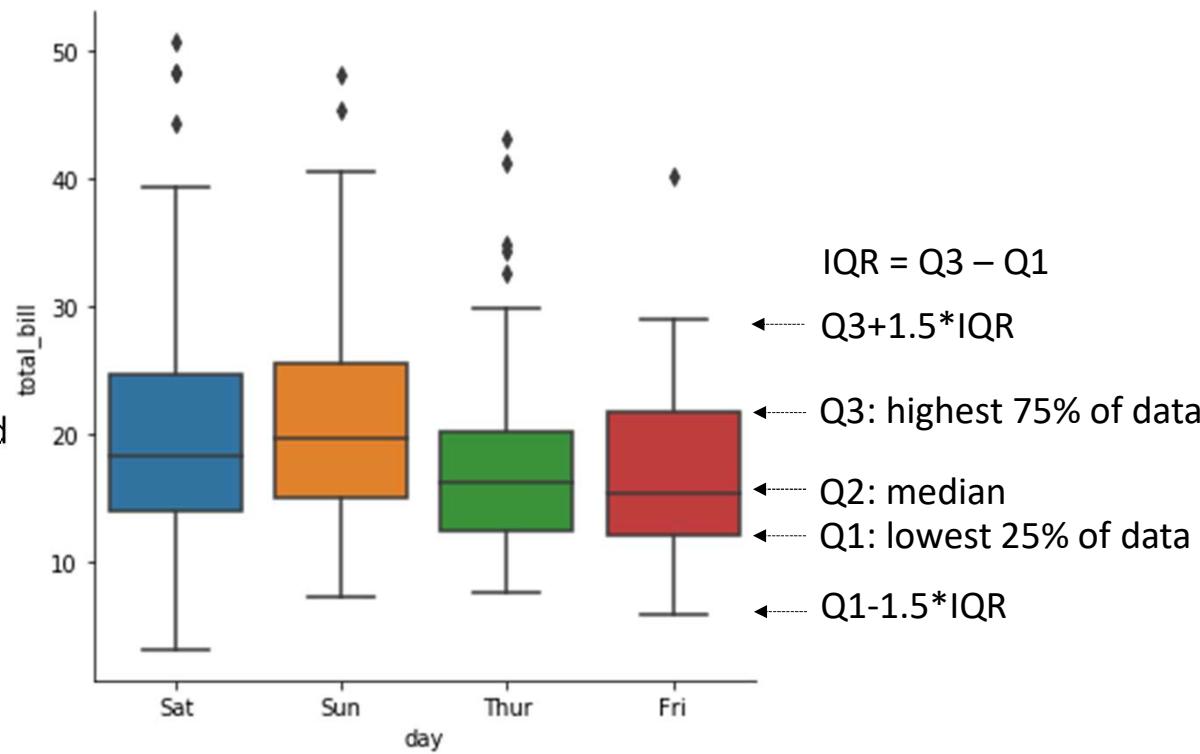
# Visualizing the distribution

- With the default scatterplots, it gets harder to see the distribution of your data when the size gets bigger.
- We can use several other plotting approaches showing the distribution information, by passing argument `kind` with:
  - box and boxen
  - violin



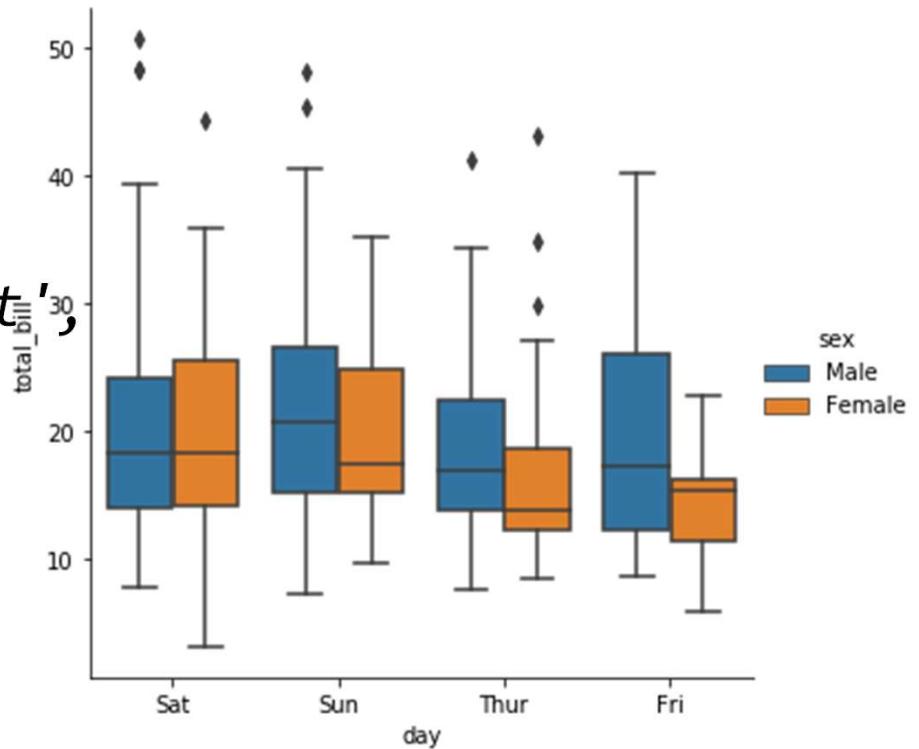
# Box-plot explained

- Box plot is used for descriptive statistics. It is created based on quartiles, and also called as **box-and-whisker plot**.
- Box is bounded by 1<sup>st</sup> and 3<sup>rd</sup> quartiles.
- Whiskers extend to 1.5 IQRs of 1st and 3<sup>rd</sup> quartiles.



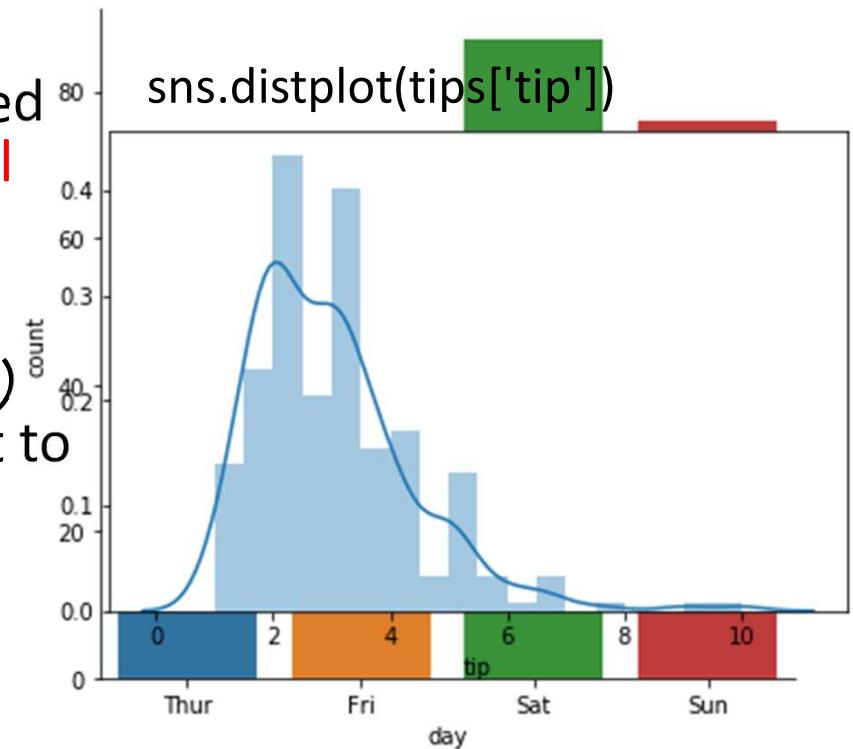
# Adding dimensions to box plot

- Box plot also supports adding third dimensions with argument `hue`.
- `sns.catplot(x="day",  
y="total_bill", order=['Sat',  
'Sun', 'Thur', 'Fri'],  
kind='box', hue= 'sex',  
data=tips)`



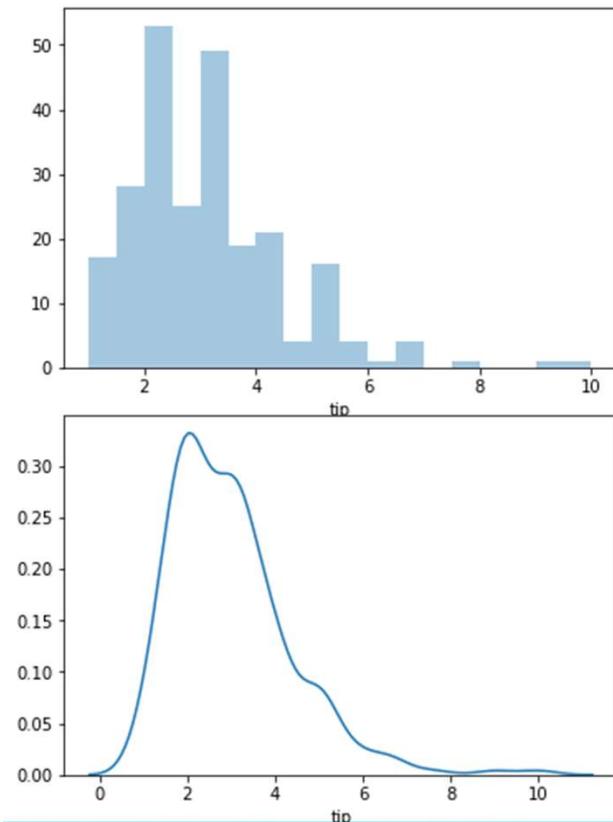
# Descriptive statistics for individual variables

- So far, we focus on the relationships between two variables. We often need to have a better idea about **individual variables**.
- We can use histogram to help us.
- It can be "achieved" with `catplot()` for categorical variables with `kind` set to "`count`";
- and `distplot()` for numeric variables.



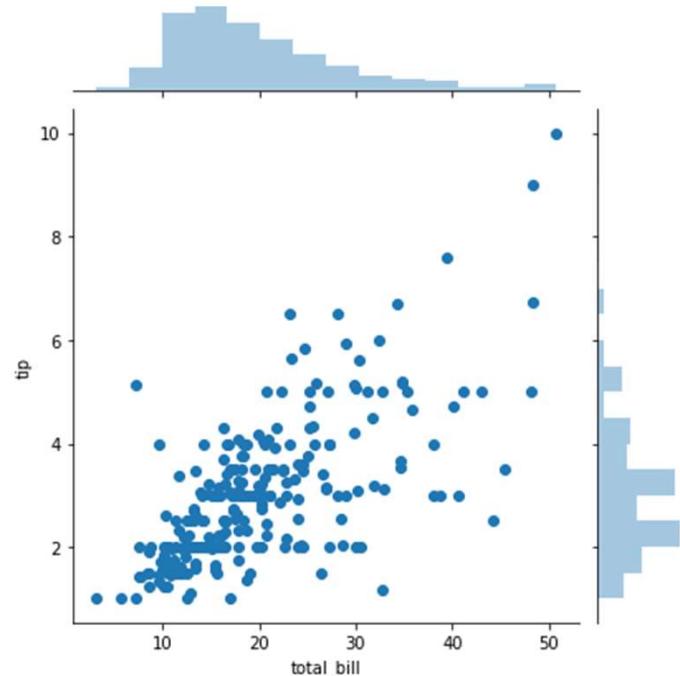
# Histogram explained

- Histogram is used to visualize univariate distributions.
- By default, it fits a kernel density estimate (KDE), which is basically a non-parametric approach to create a smoothed line based on discrete data.
- You may turn on and off histogram and KDE by setting arguments:
  - `kde: false or True`
  - `hist: False or True`



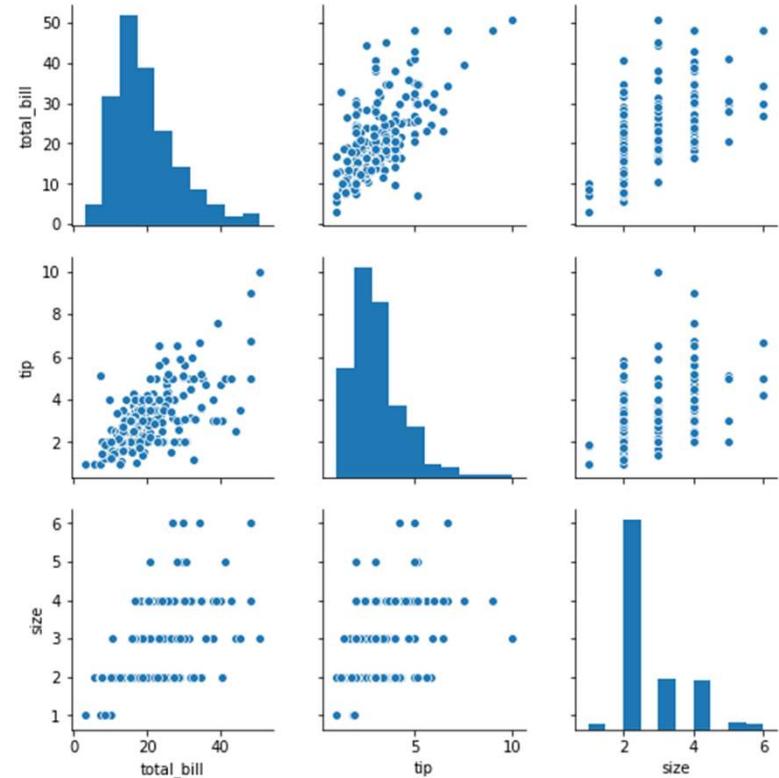
# Joining histogram and scatter plots

- You can combine histogram (one variable) and scatter plots (two variables) together to give a comprehensive view.
- This can be done with *jointplot()*;
- *sns.jointplot(x="total\_bill", y="tip", data=tips)*

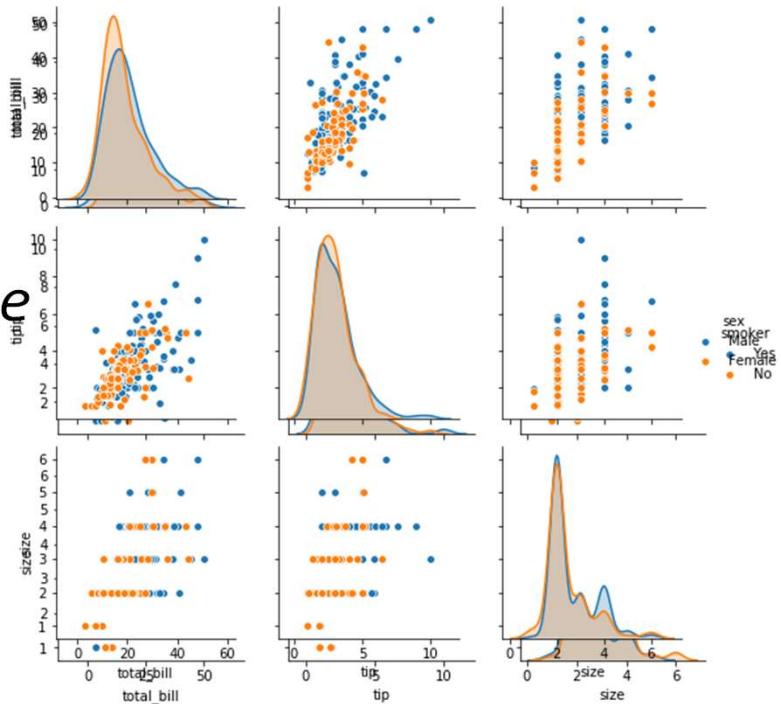


# Visualizing pairwise correlations

- A very handy function `pairplot()` can be used to visualize the **pairwise correlations** for a quick overview.
- `sns.pairplot(tips)`
- When you have too many dimensions in your dataset, you may specify the dimensions in a **list** and pass to argument **vars**:



- `pairplot()` only plots **continuous** (numeric) variables. You may pass **categorical** variable with argument `hue` to group the data.
- `sns.pairplot(tips, hue='smoker')`



# Exercise

- [Iris flower data set](#) is a famous data set originally used by biologist Ronald Fisher in 1936. It is about three different species of Iris flowers. It becomes a classic dataset for practicing data science techniques and is provided as built-in dataset in many Python libraries.
- Import the Iris dataset and explore factors related to the speciation.

```
iris = sns.load_dataset("iris")
```

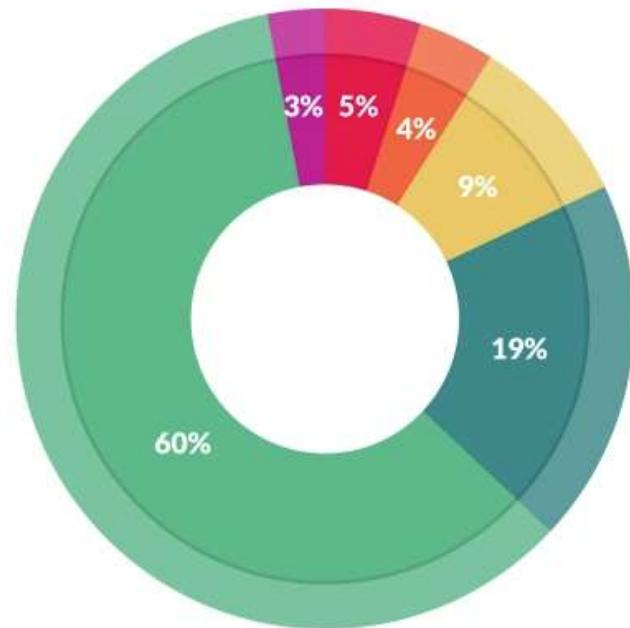


# Data Wrangling

# Data cleaning and processing

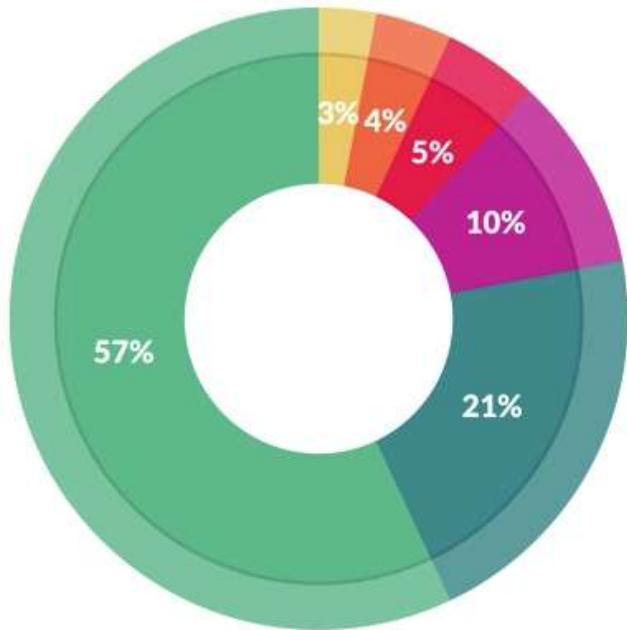
- Data wrangling is an extremely important yet time consuming step that to process and clean your data in order to transform it from **raw** data into **analysable** data.
- It may involve following operations:
  1. Importing, splitting and/or merging the datasets.
  2. Exploring your data through visualization.
  3. Generating new data through calculating or converting.
  4. Dealing with missing data.
  5. Removing unrelated data.

# What data scientists spend the most time doing



## What data scientists spend the most time doing

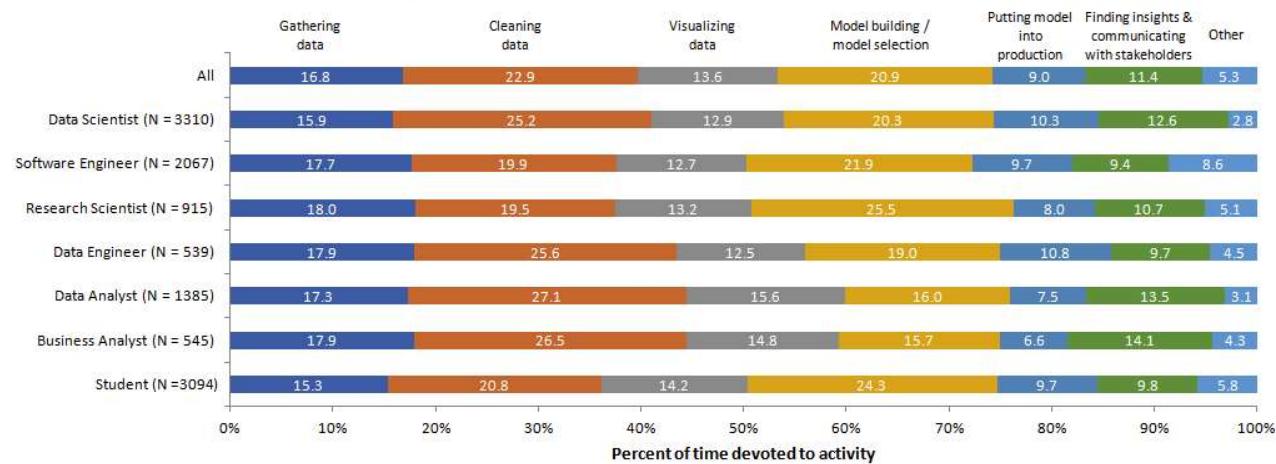
- *Building training sets: 3%*
- *Cleaning and organizing data: 60%*
- *Collecting data sets; 19%*
- *Mining data for patterns: 9%*
- *Refining algorithms: 4%*
- *Other: 5%*



### What's the least enjoyable part of data science?

- *Building training sets: 10%*
- *Cleaning and organizing data: 57%*
- *Collecting data sets: 21%*
- *Mining data for patterns: 3%*
- *Refining algorithms: 4%*
- *Other: 5%*

**During a typical data science project at work or school, approximately what proportion of your time is devoted to the following?**



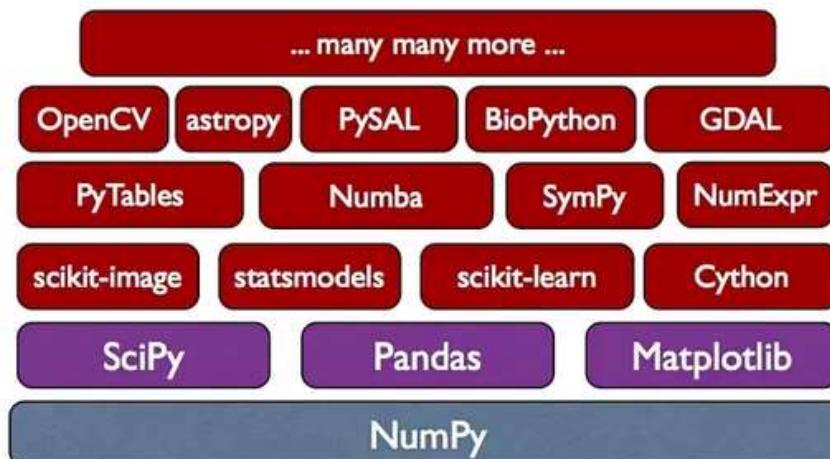
Note: Data are from the 2018 Kaggle ML and Data Science Survey. You can learn more about the study here: <http://www.kaggle.com/kaggle/kaggle-survey-2018>. A total of 23859 respondents completed the survey; the percentages in the graph are based on a total of 15937 respondents who provided an answer to this question. Only selected job titles are presented.



Copyright 2019 Business Over Broadway

# NumPy

- NumPy stands for numerical Python.
- Foundation library for scientific computing in Python



<https://www.quora.com/What-is-the-relationship-among-NumPy-SciPy-Pandas-and-Skikit-learn-and-when-should-I-use-each-one-of-them>

# Advantages over built-in functions

- Ndarray: A **multidimensional array** much faster and more efficient than those provided by the basic package of Python.
- Element-wise computation: A set of functions for performing this type of **calculation with arrays** and mathematical operations between arrays.
- Reading-writing datasets: A set of tools for reading and writing data stored in the hard disk.
- Integration with other languages such as C, C++, and FORTRAN: A set of tools to integrate code developed with these programming languages.

# Installation

- NumPy should be installed as part of the Anaconda installation. If not or in a new environment:

conda install numpy

Pip install numpy

- It is pre-installed in Colab.

# Quick recall of Python data types

- Number
- String
- List
- Tuple
- Dictionary

# NumPy array

- The NumPy library is based on one main object: ndarray (which stands for N-dimensional array).
- A ndarray is created by the *array()* function.

```
>>> a = np.array([1, 2, 3]) # list as argument  
>>> a  
array([1, 2, 3])  
>>> type(a)  
<type 'numpy.ndarray'>
```

- ndarray is **homogeneous**: consisted of data in the same date type.

# NumPy data type dtype

- `bool_` Boolean (True or False) stored as a byte
- `int_` Default integer type (same as C long; normally either int64 or int32)
- `intc` Identical to C int (normally int32 or int64)
- `intp` Integer used for indexing (same as C ssize\_t; normally either int32 or int64)
- `int8` Byte (-128 to 127)
- `int16` Integer (-32768 to 32767)
- `int32` Integer (-2147483648 to 2147483647)
- `int64` Integer (-9223372036854775808 to 9223372036854775807)
- `uint8` Unsigned integer (0 to 255)
- `uint16` Unsigned integer (0 to 65535)
- `uint32` Unsigned integer (0 to 4294967295)
- `uint64` Unsigned integer (0 to 18446744073709551615)
- `float_` Shorthand for float64.
- `float16` Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
- `float32` Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
- `float64` Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
- `complex_` Shorthand for complex128.
- `complex64` Complex number, represented by two 32-bit floats
- `complex128` Complex number, represented by two 64-bit floats

Python

`int`  
`bool`  
`float`  
`complex`  
`bytes`  
`str`

Numpy

`int_`  
`bool_`  
`float_`  
`cfloat`  
`bytes_`  
`unicode_`

```
>>> a.dtype  
dtype('int64')
```

```
>>> a.dtype  
dtype('int64')  
>>>b = np.array(['a', 2, 3])  
dtype('<U1')
```

If you mix some strings with the numbers then all of the elements will get converted into a **string** type and we won't be able to perform most of the numpy operations on that array.

# Creating a Numpy Array

- A numpy array can be created in three ways:

## 1. From Python **list** or **tuple**:

```
>>>larray = np.array([1,2,3])
>>>tarray = np.array((1,2,3))
>>>larray
array([1, 2, 3])
>>>tarray
array([1, 2, 3])
```

```
>>>marray = np.array([[1,2,3],
                     [2,3,4],
                     [3,4,5]])
>>>marray
array([[1, 2, 3],
       [2, 3, 4],
       [3, 4, 5]])
```

## 2. Using numpy functions:

❖ *arange(start, stop, step)*, similar to *range()*.

```
>>>narray = np.arange(1,10,2)
```

```
>>>narray
```

```
array([1, 3, 5, 7, 9])
```

❖ other functions to create special forms of array.

### 3. Reading data files.

import data in text (csv) file into numpy array with *getfromtxt()* and *Loadtxt()*.

- *getfromtxt()* gives additional options if you have missing data.
- *Loadtxt()* is equivalent to *getfromtxt()* if there is no missing data.

```
>>>farray = np.getfromtxt('data.csv')
```

# Advantages of numpy array

- python list/tuple is **dynamic** typed, it can be mixed with different types of data and the type is **not pre-defined**.
- numpy array is **statically typed** and **pre-defined**. It can leverage complied language like C to improve the computing efficiency.
- numpy array supports **complex mathematical calculations**, such as matrix and dot multiplication, which can improve efficiency significantly.

## Case: multiply numbers from two lists

Problem: multiply each element in a list with the corresponding element in another list of the same length.

Python loop statements

```
c = []
for i in range(len(a)):
    c.append(a[i]*b[i])
```

Numpy operation

```
c = a * b
```

# Shape of NumPy array

- The `shape` property is used to get the current shape of an array. It returns **tuple** of array dimensions

```
>>>larray.shape
```

(3,) #a has only one dimension and three elements in the first dimension.

```
>>>marray.shape
```

(3, 3) # c has two dimensions and two elements in the first dimension, 3 elements in the second dimension. In other words, it has three one-dimensional elements, each with three elements.

# Exercise

```
>>>c = np.array([[[ 1,  2,  3,  4],  
                 [ 5,  6,  7,  8],  
                 [ 9, 10, 11, 12]],  
                [[13, 14, 15, 16],  
                 [17, 18, 19, 20],  
                 [21, 22, 23, 24]]])  
  
>>>c.shape  
(2,3,4)
```

# Reshape your array

- Numpy array can be reshaped.

```
>>>c.reshape(3,2,4)
array([[[ 1,  2,  3,  4], [ 5,  6,  7,  8]],
       [[ 9, 10, 11, 12], [13, 14, 15, 16]],
       [[17, 18, 19, 20], [21, 22, 23, 24]]])
```

```
>>>c.reshape(4,3,2)
array([[[ 1,  2], [ 3,  4], [ 5,  6]],
       [[ 7,  8], [ 9, 10], [11, 12]],
       [[13, 14], [15, 16], [17, 18]],
       [[19, 20], [21, 22], [23, 24]]])
```

reshape does not change  
the original array. It only  
returns the result of  
reshaped array.

# Special reshape

- You can easily transpose a 2-D array with `T` or `transpose()`.

```
c = np.arange(1,25).shape(4,6)  
c = c.T  
C = c.transpose()
```

- You can also flatten multi-dimensional array into one dimension array with `ravel()`.

```
c = c.ravel()
```

- You can directly change the shape of array by `resize()`.

```
c.resize(2,3,4)
```

# Array stacking

- Multiple arrays can be stacked, vertically and horizontally.

```
a1 = np.array([[1,2],[3,4]])
a2 = np.array([[5,6],[7,8]])
a3 = np.vstack((a1,a2))
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
a4 = np.hstack((a1,a2))
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

## More attributes of numpy array

- `ndim` returns the **total number of dimensions** of the array.

```
>>>c.ndim
```

3

- `size` returns the **total number of elements** in the array.

```
>>>c.size
```

24

# More functions to create numpy array

- The function `zeros()` creates an array full of **zeros**,
- The function `ones()` creates an array full of **ones**,
- The function `empty()` creates an array whose initial content is **random** and depends on the state of the memory.
- You only need to specify `shape` of array as a **list or tuple** to be generated.

```
np.zeros([2,3])
```

- You can provide data type, `dtype`, as arguments to these functions.

# Example

- Create a 3 by 2 matrix with all zeros with data type int16.

```
>>>zarray = np.zeros((3,2), dtype=np.int16)
```

Create a 2 by 3 by 4 array with all ones with data type float64.

```
>>>Create an empty 3 by 3 array with data type
```

```
>>>earray = np.empty((3,3))
```

# Index your array

- Similar to Python **nested list**, you can index and slice your array.
- For multi-dimensional array, use comma to separate index for each dimension.

```
>>>a = np.arange(1,25).reshape(2,3,4)
array([[[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]],
      [[13, 14, 15, 16],
       [17, 18, 19, 20],
       [21, 22, 23, 24]]])
>>>a[1,2,3]
24
```

- If index has been specified for all dimensions, a data value will be returned, otherwise a NumPy array will be returned.

```
>>>a[1,2]
array([21, 22, 23, 24])
>>>a[1]
array([[13, 14, 15, 16],
       [17, 18, 19, 20],
       [21, 22, 23, 24]])
```

# Slice the array

- Slice 1-D array with [start:end:step]

```
>>>a = np.array([1,2,3,4,5,6,7,8])
```

```
>>>a[1:5]
```

```
array([2, 3, 4, 5])
```

```
>>>a[1:6:2]
```

```
array([2, 4, 6])
```

# Slice multi-dimensional array

- Use comma separating slicing on each dimension. When fewer indices are provided than the number of dimensions, the missing indices are considered **complete** slices

```
>>>a[0:2,0:2,0:2]
array([[[ 1,  2], [ 5,  6]],
       [[13, 14], [17, 18]]])
>>>a[0:2,0:2]
array([[[ 1,  2,  3,  4], [ 5,  6,  7,  8]],
       [[13, 14, 15, 16], [17, 18, 19, 20]]])
>>>a[0:2]
array([[[ 1,  2,  3,  4], [ 5,  6,  7,  8], [ 9, 10, 11, 12]],
       [[13, 14, 15, 16], [17, 18, 19, 20], [21, 22, 23, 24]]])
```

## Index and slice your array

```
>>>a[1,1:3,3]  
array([20, 24])
```

With high dimensional array, it may get harder to index. We can use three dots `...` to indicate complete slices. For example, `x` is a 5-D array.

- `x[1,2,...]` is equivalent to `x[1,2,:,:,:]`,
- `x[...,3]` to `x[:,::,:,3]` and
- `x[4,...,5,:]` to `x[4,:,:5,:]`.

# Index tricks

- Using array as index.

```
>>> a = np.arange(10)
>>> i = np.array([ 1,1,4,7,5 ])
>>> a[i] # the elements of a at the positions i
array([ 1,  1,  4,  7,  5])
>>> j = np.array( [ [ 3, 4], [ 9, 7 ] ] )
>>> a[j]
array([[ 3,  4], [9,  7]]) # return an array with the same
shape as j
```

# Basic operations

- Arithmetic operators on arrays apply **elementwise**.

```
>>>a = np.array([1,2,3])
>>>b = np.array([4,5,6])
>>>a + b # array([5, 7, 9])
>>>a - b # array([-3, -3, -3])
>>>a * b # array([ 4, 10, 18])
>>>a / b # array([0.25, 0.4 , 0.5 ])
>>>a ** b # array([ 1, 32, 729], dtype=int32)
>>>a < 2 # array([ True, False, False])
```

\* is not for matrix  
product in NumPy.

# Matrix product

$$\begin{pmatrix} A1 & A2 \\ A3 & A4 \end{pmatrix} \times \begin{pmatrix} B1 & B2 \\ B3 & B4 \end{pmatrix} = \begin{pmatrix} A1*B1 & A1*B2 \\ +A2*B3 & +A2*B4 \\ A3*B1 & A3*B2 \\ +A4*B3 & +A4*B4 \end{pmatrix}$$

Matrix product can be performed with `@` or `dot()`.

```
>>>a = np.array([[1,2],[3,4]])  
>>>b = np.array ([[5,6],[7,8]])  
>>>a @ b # array([[19, 22], [43, 50]])  
>>>a.dot(b) # array([[19, 22], [43, 50]])
```

# Matrix product

- Make sure you have same number of elements in matching row and column.

```
>>>a = np.array([[1,2,3],[3,4,5]])  
>>>b = np.array ([[5,6],[7,8]])  
>>>a @ b # ValueError: shapes (2,3) and (2,2) not aligned: 3 (dim 1) != 2 (dim 0)
```

- The number of columns ( $2^{\text{nd}}$  dimension) in the first matrix should equal to the number of rows ( $1^{\text{st}}$  dimension) in the second matrix

```
>>>c = np.array ([[5,6],[7,8],[9,10]]) #shape(3,2)  
>>>a @ b # array([[46,52], [88,100]])
```

# Computation comparison

```
rng = np.random.RandomState(42)
x = rng.rand(100000) #(100000 ,1)
y = rng.rand(100000)
%timeit x + y
2.43 ms ± 52.3 µs per loop
```

```
%timeit results = [xi + yi for
xi, yi in zip(x, y)]
result = []
for xi, yi in zip(x,y):
    results.append(xi+yi)
181 ms ± 2.43 ms per loop
```

## For more information

- <https://docs.scipy.org/doc/numpy/user/quickstart.html>

# Pandas

- Pandas is a ... name Pandas (from Wiki),
- It depends ...
- To some extent dealing with functionality
- It is probab ...



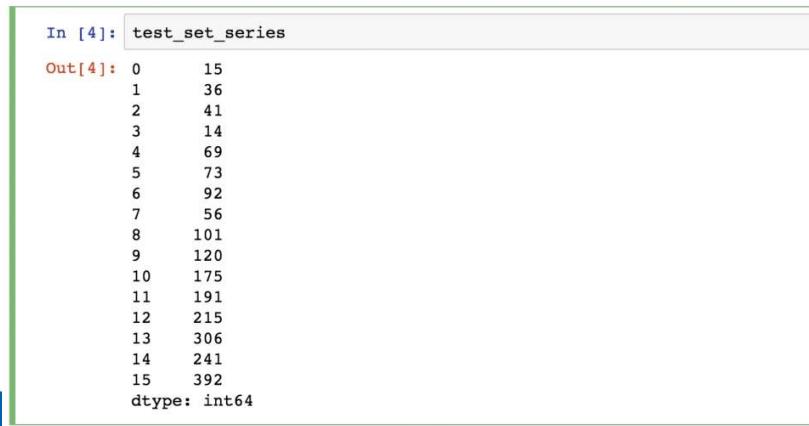
analysis. The library ("Panel Data" and Matplotlib. The library mostly deals with data manipulation and analysis. The library for Python.

# Data structure

- There are two types of data structures in pandas: **Series** and **DataFrames**.
- **Series**: one dimensional data structure (“a one dimensional ndarray”), and for every value it holds a unique **index**.
- **DataFrame**: a **two dimensional** data structure – basically a table with rows and columns. The columns have names and the rows have **indexes**.

1. A series can be created using pandas function Series with python **list** or numpy **1-D array** as the argument. By default, each item will receive an numeric index label starting from 0.

```
>>>s1 = pd.Series([1,2,3])  
>>>s2 = pd.Series(np.array([1,2,3,4,5]))
```



In [4]: test\_set\_series

Out[4]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	15	36	41	14	69	73	92	56	101	120	175	191	215	306	241	392

dtype: int64

# Manually creating a Series data

1. An explicit index can also be specified when creating the series by providing the **index** with a **list** as the second argument. This is often called **label**.

```
>>>s3 = pd.Series([1,2,3,'a','b','c'],  
                  index=['A','B','C','D','E','F'])
```

2. When a dictionary is provided as the argument, the **key** will be used as the index.

```
>>> s4 = pd.Series({'A':1,'B':2,'C':3})
```

3. Each index label needs to be unique?

# Indexing and slicing Series Data

1. Data in the series can be accessed similar to that in a Python list **when having the default numeric index.**

```
s2[2]
```

```
s2[:2] # return a series data.
```

2. Data in the series can be accessed similar to that in a Python dictionary when having specified index label.

```
s3['A']
```

3. You can retrieve multiple data by providing a **list** of "keys"/labels.

```
s3[['A', 'B', 'C']]
```

# Dataframe

- A **dataframe** has labeled axes (rows and columns) and can be created by pandas function: *DataFrame()*

```
In [12]: big_table
```

```
Out[12]:
```

	user_id	phone_type	source	free	super
0	1000001	android	invite_a_friend	5.0	0.0
1	1000002	ios	invite_a_friend	4.0	0.0
2	1000003	error	invite_a_friend	37.0	0.0
3	1000004	error	invite_a_friend	0.0	0.0
4	1000005	ios	invite_a_friend	6.0	0.0

# Create DataFrame from a dictionary

- You can create a DataFrame from **dictionary** of **narrays/lists/series**. Keys will be used as the **column labels** by default. Values become the columns corresponding to the key.

```
>>> d1 = pd.DataFrame({ 'A' : [1,2,3] , 'B' : [2,3,4] })
```

- You may also specify index label for the rows.

```
>>> d2 = pd.DataFrame({ 'A' : [1,2,3] , 'B' : [2,3,4] }, index=['X','Y','Z'])
```

```
>>> d3 = pd.DataFrame({ 'A':np.array([1,2,3]) ,  
                      'B':[2,3,4] }, index=['X','Y','Z'])
```

```
>>> d4 = pd.DataFrame({ 'A' : [1,2,3] , 'B':s1}) #s1 is a ndarray
```

# Create DataFrame from a dictionary

- Note: Items in the dictionary must have the **same length** unless they are all series.

```
>>>d5 = pd.DataFrame({'A' : [1,2,3], 'B' :[2,3,4,5]}) #error  
>>>d6 = pd.DataFrame({'A' : s1, 'B' :[1,2]}) #error
```

- When series have different length, Python will try to match their index to create the dataframe and NaN (Not a Number) is appended in missing areas.

```
>>>d7 = pd.DataFrame({'A' : s1, 'B' :s2}) #using default numeric  
index  
>>>d8 = pd.DataFrame({'A' : pd.Series([1, 2, 3], index=['a',  
'b', 'c']), 'B' : pd.Series([1, 2, 3, 4], index=['b', 'c',  
'd', 'e'])})#using specified index
```

# Create DataFrame from a list

- A DataFrame can be created using a single list or a list of lists.

```
>>> d9 = pd.DataFrame([1,2,3,'a','b','c']) #compare with s1.
```

```
>>> d10 = pd.DataFrame([[1,2,3],[2,3,4],[3,4,5]])
```

- Numeric labels will be created for row and column by default. You can also specify the labels for `columns` and `index` (row).

```
>>> d11 = pd.DataFrame([[1,2,3],[2,3,4],[3,4,5]],columns=['A','B','C'])
```

```
>>> d12=pd.DataFrame([[1,2,3],[2,3,4],[3,4,5]], columns=['A','B','C'],  
index=['X','Y','Z'])
```

# Create DataFrame from a list

- You can create a DataFrame from a **list of dictionaries**. Keys will be used as the **column** labels by default.

```
>>>d13 = pd.DataFrame([{'a': 1, 'b': 2}, {'a': 5, 'b': 10}])  
>>>d14 = pd.DataFrame([{'a': 1, 'b': 2}, {'a': 5, 'b': 10}], index=['A', 'B'])
```

- Each item in the list is like a **row** in a table. Items in the list can have different length.

# Exercise

- Create a DataFrame with shape(1,3)

# List of elements with different lengths

- When no specific column label is provided, Python will match the default labels (number index or keys) to create the dataframe and **NaN** is appended in missing areas.

```
>>> d15 = pd.DataFrame([[1,2],[2,3],[3,4,5]])  
>>> d16= pd.DataFrame([{'a': 1, 'b': 2},{'b': 5, 'c': 10,'d':15}])
```

- When column labels are specified, Python will create DataFrame based on the column labels and try to match keys with the labels. Values with non-match keys will be **ignored**.

```
>>>d17 = pd.DataFrame([{'a': 1, 'b': 2},{'b': 5, 'c':10,'d':15}], columns=['b','d','e'])
```

# Create DataFrame from files

- Pandas can read data directly from a wide range of file formats, such as csv, Excel, JSON, SQL database, Stata, SAS, etc. We will focus on csv files in this class.
- Use *read\_csv()* function. **Filename** is the only required argument.

```
df_titan = pd.read_csv('titanic_train.csv')
```

- ❖ Many optional arguments can be passed when importing data.
- ❖ [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read\\_csv.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html)

# Key parameters of read\_csv()

- **delimiter**: comma by default, set when necessary.
- **header**: row to be used as the column headers, and the start of data. 0 by default. Set to None if no header.

```
df_titan = pd.read_csv('titanic_train.csv',header=None)
```

- **names**: a list of column names to be used instead header.

```
df_titan = pd.read_csv('titanic_train.csv',header=None,names=[1,2])
```

- **index\_col**: specify a column to be used for row index.

```
df_titan = pd.read_csv('titanic_train.csv',index_col=0)
```

Row index can also be specified with column name

```
df_titan = pd.read_csv('titanic_train.csv',index_col='PassengerId')
```

## Key parameters of read\_csv()

- **usecols**: import selected columns by passing a **list** of column index or names.

```
columns = [2,3,4] #columns = ['Pclass', 'Name', 'Sex']
```

```
df_titan = pd.read_csv('titanic_train.csv', usecols=columns)
```

- **skiprows**: specify the number of first n rows to skip.
- **nrows**: specify the total number of rows to read. Useful when reading large files.

```
df_titan = pd.read_csv('titanic_train.csv', skiprows=3, nrows=10)
```

## Key parameters of `read_csv()`

- `na_values`: list of strings to be treated as `NaN`. Most common ones can be detected automatically, such as #N/A, n/a, null, etc.

```
missing = ['not available', 'missing']
df_titan = pd.read_csv('titanic_train.csv', na_values=missing)
```

# Column selection and deletion

- Column selection using the column label:

```
>>> print(df_titan['Age']) #column label as the key, return a series  
>>> print(df_titan[['Age']]) #return a dataframe
```

- Add new column with label, similar as adding new item to a dictionary:

```
>>> df_titan['f'] = pd.Series([10,10]) #series/list/narray
```

- New column can be added by calculating existing columns:

```
>>> df_titan['g'] = df_titan['b'] + df_titan['f'] #NaN if one cell is NaN.
```

- del to delete a column:

```
>>> del df_titan['g']
```

# Row Selection

- Row selection by passing row **labels** to `loc[]` method. The row will be returned as a **series** or a **dataframe**:

```
>>> df_titan.loc[1] #column labels will be used as row index.
```

- Multiple rows can be selected:

```
>>> df_titan.loc[['a','c','e']] #a list of indexes/labels,  
returns a dataframe
```

```
>>> df_titan.loc['a':'c'] #slice, both start and end included.
```

- Column labels can be provided to filter the results:

```
>>> df_titan.loc[['a','c','e'],'A']
```

- Select rows with Boolean list indicating whether to be selected:

```
>>> df_titan.loc[[True, False, False, True, False]] # same  
length as row.
```

- Select rows with Boolean expression:

```
>>> df_titan.loc[df8['B'] > 2]  
>>> df_titan.loc[df8['B'] > 2, 'A'] # only display column A
```

# Row Selection

- Select rows by passing integer position (index) to method `iloc[]`.

```
>>> df_titan.iloc[1] #implicit numeric index starting from 0.
```

- Note: `df8.loc[1]` differs from `df8.iloc[1]`

```
>>> df_titan.iloc[0:2]
```

## Add rows

- Add new rows with `append()` method at the end of a dataframe.

```
>>> df_titan.append([99,99]) #existing dataframe not updated.
```

```
>>> df_titan.append(pd.DataFrame([99,99]))
```

- Python will align matching columns.

```
>>> df_titan.append(pd.DataFrame([99,99],columns=['A','B']))
```

```
>>> df_titan.append(pd.DataFrame([[99,99]],columns=['A','B']))
```

- You can reset row labels by setting `ignore_index=True` (default is False).

- `df_titan.append(pd.DataFrame([[99,99]],columns=['A','B']),  
ignore_index=True )`

# Row deletion

- You can remove rows by `drop()` method with a **list** of labels.

```
df_titan.drop([4]) #same as df8.drop(4)
```

```
df_titan.drop([0:2]) #error [0:2] is not a list. Alternatively, df8 = df8[2:].
```

```
df_titan.drop([0,1,2])
```

```
df_titan.drop(range(3)) #existing dataframe not updated
```

```
df_titan = df8.drop([4])
```

```
df_titan.drop([5], inplace=True) # existing dataframe updated.
```

# Important dataframe attributes

- `.index` returns a "list" of **row indexes** (numeric positions rather than labels).
- `df_titan.index`
- `df_titan.loc[df_titan['B'] > 2].index`
- Very handy to remove rows based on condition.
- `df_titan.drop(df_titan.loc[df_titan['B'] > 2].index)`

# Scalar operation

- Basic arithmetic and Boolean operations with scalar data are **element-wise**.

- `df_titan *2 #broadcasting`
- `df_titan.add(2)`
- `df_titan >2`
- `df_titan['B'] *2`

Python Operator	Pandas Method(s)
+	<code>add()</code>
-	<code>sub(), subtract()</code>
*	<code>mul(), multiply()</code>
/	<code>truediv(), div(), divide()</code>
//	<code>floordiv()</code>
%	<code>mod()</code>
**	<code>pow()</code>

# More operations

- Arithmetic and Boolean operations with another list or Series will be performed based on **matching** labels (columns).

- `d10 = pd.DataFrame( [[1,2,3], [3,4,5], [5,6,7]] )`
- `d10 - [1,2,3]` #default to compare by column.
- `d10 > [3,3,3]`
- `d10 - [1,2]` #error, different length
- `d10 - pd.Series([1,2])` # NaN for no-match column.
- `d8 - pd.Series([1,2])`

- For operations by row, you need to specify `axis` to be "index".
  - `d10.sub(pd.Series([1,2,3]),axis='index')`
  - `d10.sub(pd.Series([1,2,3]),axis='index') # NaN for no-match rows.`
  - `d10.mul(pd.Series([1,2,3],index=[1,2,3]),axis='index') # NaN for no-match rows.`

# Data processing

# Explorer Dataset

- Dimensions of the dataset.

```
df_titan.shape # quick view of the dimensions.
```

```
df_titan.size # total number of measurements in your data.
```

```
df_titan.info() # count and datatype for each column
```

- Preview your data with head(n) and tail(n), n is 5 by default.

```
df8.head(3) # first 3 rows; tail(3) for last 3 rows.
```

# Descriptive statistics

- The **describe()** method computes a summary (NaN will be excluded) of statistics pertaining to the DataFrame columns. Summary is also a **DataFrame**.

```
df_titan.describe()
```

- You may pass proper value to argument **include**:
  - **number** – default, only summarizes Numeric columns.
  - **object** – only summarizes **String** columns.
  - **all** – Summarizes all columns together (Should not pass it as a list value).

```
df_titan.describe(include='all')
```

# Descriptive statistics

- You can get descriptive measurements using methods like: `sum()`,`min()`,`max()`,`mean()`,`count()`,`median()`,`std()`, and more. All measurements are returned as `Series`.

```
df_titan.mean()
```

- By default, these statistics are for each column. You can get row-based statistics by specifying `axis=1`.

```
df_titan.mean(axis=1) # default axis = 0
```

- To get the unique values and the corresponding counts for a column:

```
df_titan['Pclass'].value_counts() #only works for individual column.
```

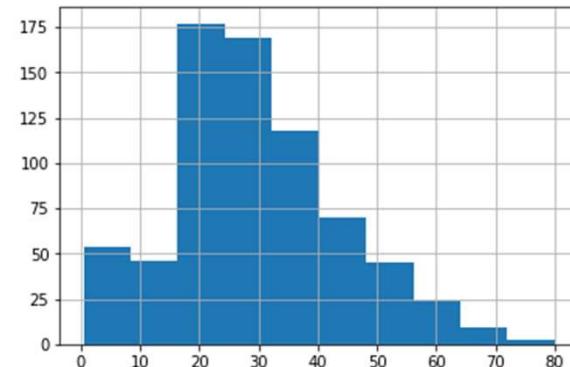
# Exploring your data with simple visualization

- Pandas offer some functions for basic plotting.
- Histogram

```
df_titan.hist()
```

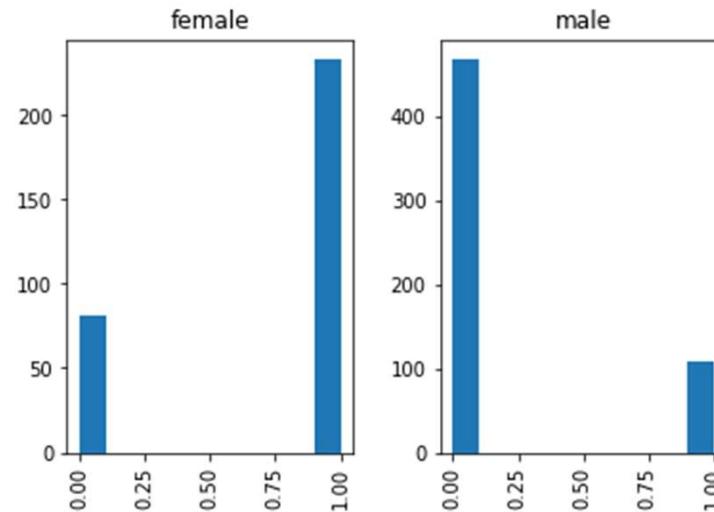
```
df_titan.hist('Age')
```

- Line
- Pie
- <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html>



- Instead of showing a single distribution, you can split based on another column.

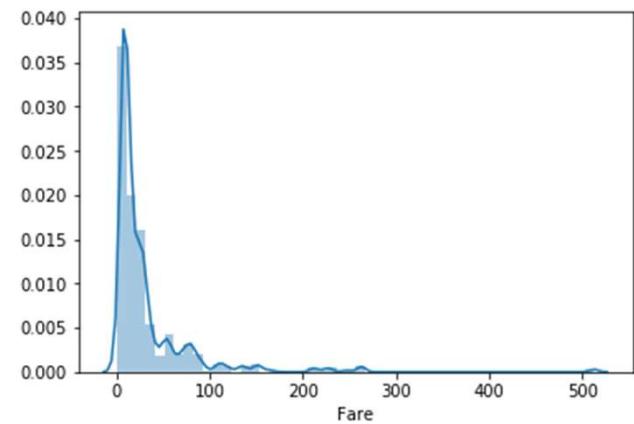
```
df_titan.hist("Survived", by="Sex")
```

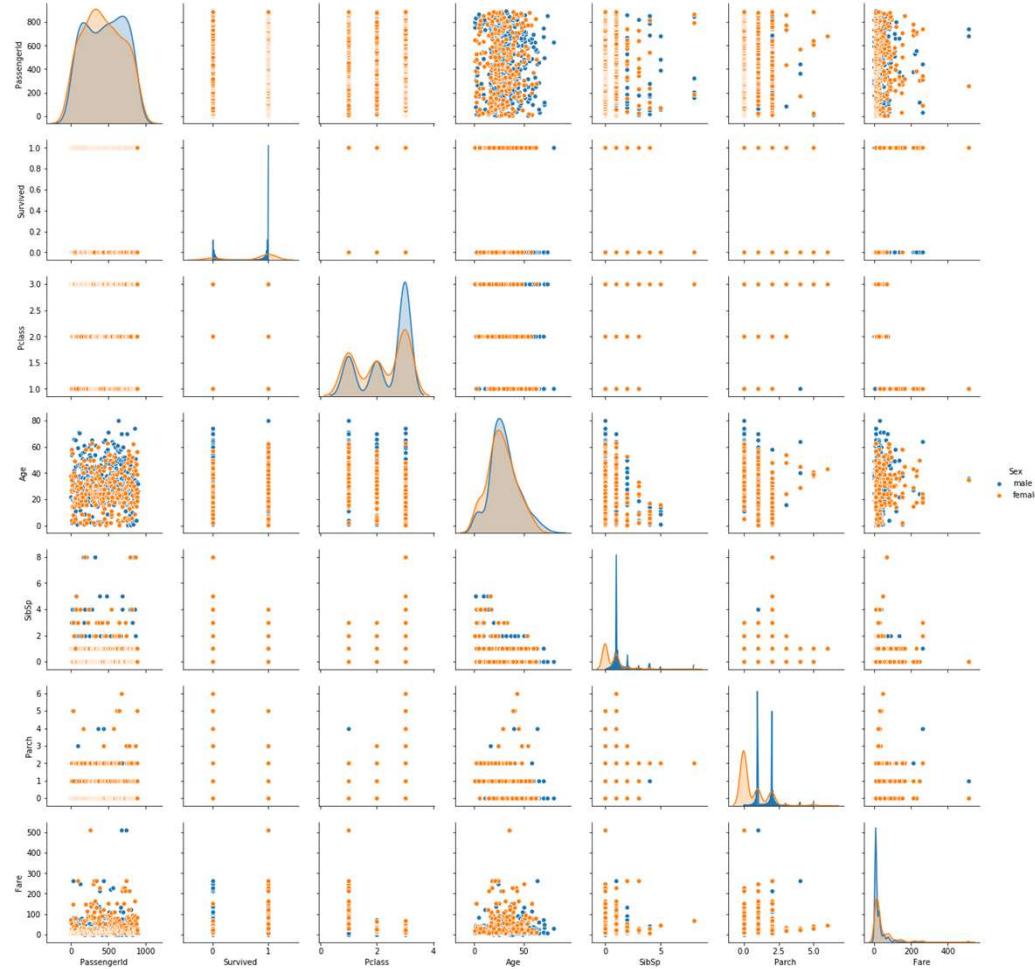


# Visualization with Seaborn

- Pandas dataframe can be used directly by Seaborn as dataset, if there is no NaN value.

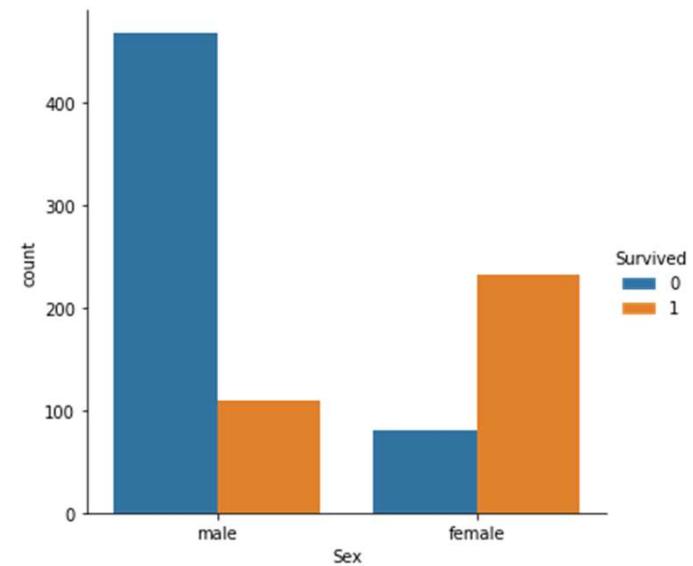
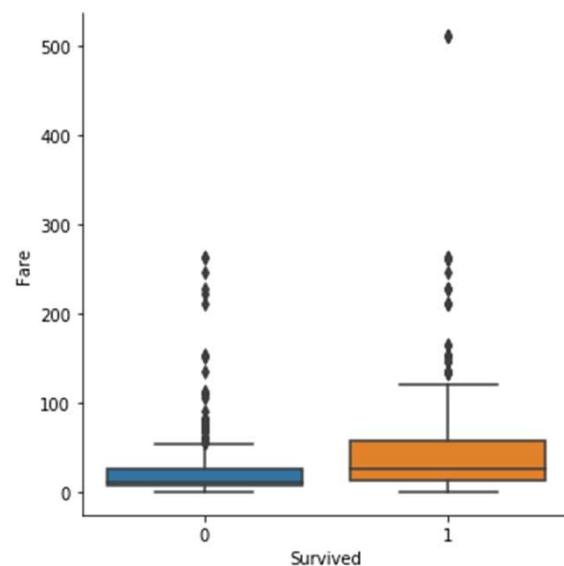
```
import seaborn as sns  
sns.distplot(df_titan['Age'])  
sns.distplot(df_titan['Fare'])  
sns.distplot(df_titan['Fare'],  
hue='Sex')
```





# Exercise

- Can you do a quick check if people paying higher fare was more likely to survive? Was female passenger more likely to survive?



# Detect missing data

In most cases, you will have missing data issue in your dataset.

Check if there is any missing data

- `df_titan.describe(include='all')` #missing data in Age and Cabin.
- `df_titan.isnull().sum()` # another trick to find out missing data.

# Reasons for missing data

1. Missing Completely at Random(MCAR): The missingness has nothing to do with any other factors.
  - ❖ Age is missing due to neglect.
2. Missing at Random(MAR): The missingness is caused by other items been measured but has nothing to do with the its own measurement.
  - ❖ Age is missing because female don't like to report their ages.
3. Missing Not at Random (MNAR): the missingness is caused by its own measurement.
  - ❖ Age is missing because elder people don't like to report their ages.

# Strategies to deal with missing data

- Delete data with missing value, with **CAUTION**.
- Removing rows with missing data by using dropna().

```
df_titan.dropna() # old dataframe will not be replaced automatically.
```

- Removing columns with missing data by specifying axis=1

```
df_titan.dropna(axis=1) # age and cabin dropped.
```

- Set threshold to remove.

```
df_titan.dropna(thresh=11) # keep rows with at least 11 non-missing data. i.e. rows missing both age and cabin get dropped.
```

# Strategies to deal with missing data

- Impute missing data. Very tricky and a lot of consideration.

<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3668100/>

1. Impute with a constant with `fillna()`.

```
df_titan.fillna(1) #fill all NaN with 1.
```

```
df_titan.fillna({'Age':20, 'Cabin':'B96'}) # different value for two columns.
```

2. Impute with mean/median/mode.

```
df_titan.fillna({'Age':df_titan['Age'].mean(), 'Cabin':'B96'})  
#replace with mean.
```

3. Impute with statistic estimation (will discuss later).

# Exercise

- Replace all missing ages with mean and drop the Cabin column.

```
df_titan = df_titan.fillna({'Age':df_titan['Age'].mean()}).dropna(axis=1)
```

# Data Cleaning

- Remove irrelevant columns (variables) using `drop(columns = [column names])`.

```
df_titan.drop(columns = ['Ticket']) # remove Ticket column from data.
```

- Sometimes, you may also want to remove outliers from your data.
  - For example, remove values outside 2 std of mean for normally distributed variables.

- top = df\_titan['Age'].mean() + 2\*df\_titan['Age'].std()
- bot = df\_titan['Age'].mean() - 2\*df\_titan['Age'].std()
- df\_titan[df\_titan['Age'] > top].info()
- df\_titan.drop[df\_titan[df\_titan['Age'] > top].index]

# Split multi-value columns

- Sometimes, you may want to split one column into multiple ones.
  - Datetime -> Year, Month, Date, Hour, Mins, Secs.
  - Name -> Title, First Name, Last Name.
  - Email -> Username, Domain.
- Regular expression can often do the trick.
  - Series.`str` can be used to access the values of the series as strings and apply several methods to it.
  - `extract()` is a Series.str method to capture groups in the `regex` pat as columns in a `DataFrame`.

# Example

- Extract title and last name from column Name as new columns.

```
df_titan.Name.str.extract('\s(\w+)\.')  
df_titan['Title'] = df_titan.Name.str.extract('\s(\w+)\.')  
df_titan.Name.str.extract('^(\\w+),')# NaN  
df_titan.Name.str.extract('^(\\w\\s]+),') #NaN  
df_titan.Name.str.extract('^(\\w\\s\\')+,')  
df_titan['LastName'] = df_titan.Name.str.extract('^(\\D+),')
```

# Derive and transform columns

- Sometimes, you may want to create new columns derived from existing ones or transform existing ones.
  - Normalization and standardization.
  - Log transformation.
  - Continuous to categorial.
  - Dummy variables.

# One-hot encoding

- `get_dummies(column)` is a Pandas function used to create dummy variables columns based on unique values in current column. This process is also called one-hot encoding. This function returns a `DataFrame`.
- `pd.get_dummies(df_titan['Sex'])`
- `df_titan[['Female','Male']] = pd.get_dummies(df_titan['Sex'])`

- Normalization
- `df_titan['FareNor'] = (df_titan['Fare'] - df_titan['Fare'].mean()) / df_titan['Fare'].std()`
- Log transformation
- `df_titan['FareLog'] = np.log(df_titan['Fare']) # zero division`

# Continuous to categorical

- We can group a range of continuous values into a category by using `cut(column,category,labels)` function. For `category`, you can pass three types of values:
  1. An integer: defines the number of equal-width categories.
  2. sequence of scalars : Defines the category boundaries allowing for non-uniform width.
  3. IntervalIndex : Defines the exact categories to be used.
- `pd.cut(df_titan['Age'],3) # 3 groups.`
- `pd.cut(df_titan['Age'],[0,19,61,100]) # 3 groups with boundaries.`
- `df_titan['AgeGroup'] = pd.cut(df_titan['Age'], [0,19,61,100], labels = ['Minor', 'Adult','Elder'])`

## Derived columns

- Instead of differentiating parent/children and sibling/spouse, we are only interested in family relationships.
- `df_titan['Family'] = df_titan["Parch"] + df_titan["SibSp"]`
- `df_titan.loc[df_titan['Family'] > 0, 'Family'] = 1`
- `df_titan.loc[df_titan['Family'] == 0, 'Family'] = 0`

# Final results

```
def data_clean1(dataframe):
    dataframe = pd.read_csv('titanic_train.csv', index_col='PassengerId')
    dataframe = dataframe.fillna({'Age':dataframe['Age'].mean()}).dropna(axis=1)
    dataframe = dataframe.drop(columns = ['Ticket'])
    top = dataframe['Age'].mean() + 2*dataframe['Age'].std()
    bot = dataframe['Age'].mean() - 2*dataframe['Age'].std()
    dataframe = dataframe.drop(df_titan[dataframe['Age']>top].index)
    dataframe = dataframe.drop(df_titan[dataframe['Age']<bot].index)
    dataframe['Title'] = dataframe.Name.str.extract('(\s(\w+)\. )')
    dataframe[['Female', 'Male']] = pd.get_dummies(dataframe['Sex'])
    dataframe['FareNor'] = (dataframe['Fare'] - dataframe['Fare'].mean()) / dataframe['Fare'].std()
    dataframe['AgeGroup'] = pd.cut(dataframe['Age'], [0, 19, 61, 100], labels = ['Minor', 'Adult', 'Elder'])
    dataframe['Family'] = dataframe["Parch"] + dataframe["SibSp"]
    dataframe.loc[dataframe['Family'] > 0, 'Family'] = 1
    dataframe.loc[dataframe['Family'] == 0, 'Family'] = 0
    dataframe = dataframe.drop(columns = ['SibSp', 'Parch', 'Sex'])
    return dataframe
```

2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th... <td>38.0</td> <td>71.2833</td> <td>Mrs</td> <td>1</td> <td>0</td> <td>0.795347</td> <td>Adult</td> <td>1</td>	38.0	71.2833	Mrs	1	0	0.795347	Adult	1		
3	1	3	Heikkinen, Miss. Laina	26.0	7.9250	Miss	1	0	-0.470569	Adult	0		

# Regular Expression

- A Regular Expression, or RegEx, is a sequence of characters that specifies a pattern to be searched.
- RegEx is like a mini "programming language" that embedded in Python, as well as other languages (more or less).
- For example, `\b[A-Z0-9._%+-]+\@[A-Z0-9.-]+\.\[A-Z\]{2,}\b` is a regular expression to match valid email addresses.
- Very useful for data collection, extraction and cleaning.
- But requires practice and "trial and error".

## Sets []

[ ] is used to match a single character specified in the brackets..

- [abcd]: Matches either a, b, c or d. It does not match "abcd".
- [a-d]: Matches any one alphabet from a to d.
- [a-] and [-a] | Matches a or -, because - is not being used to indicate a series of characters.
- [a-z0-9] | Matches any character from a to z and also from 0 to 9.

[^] is used to match a single character not specified in the brackets

- [^abc] matches any character that is not a, b and c.

# RegEx in Python

- Python has build-in module `re` for regular expression operation.

`re.findall(A, B)` will matches all instances of a string or an expression A in a string B and returns them in a `list`.

```
print(re.findall("o","I love python")) # ['o','o']
```

- Add `r` before string A to indicate a regular expression.

```
print(re.findall(r"[a-p]","I love python")) # ['l', 'o', 'e', 'p', 'h', 'o', 'n']
```

```
print(re.findall(r"[lop]","I love python")) # ['l', 'o', 'p', 'o']
```

```
print(re.findall(r"[o-t][v-z]","I love python")) # ['ov', 'py']
```

# Special Sequences

**\w** Matches alphanumeric characters, which means a-z, A-Z, and 0-9. It also matches the ideogram and underscore, \_.

```
print(re.findall(r"\w","I love 爱 python3")) # ['I', 'l', 'o', 'v', 'e', '爱', 'p', 'y', 't', 'h', 'o', 'n', '3']
```

**\W** matches any character not included in \w.

**\d** Matches digits, which means 0-9.

```
print(re.findall(r"\d","I love python3")) # ['3']
```

```
print(re.findall(r"\w\d","I love python3")) #['n3']
```

**\D** Matches any non-digits.

```
print(re.findall(r"\D","I love python3")) # ['I', ' ', 'l', 'o', 'v', 'e', ' ', 'p', 'y', 't', 'h', 'o', 'n']
```

# Example

- Find all WBS student id in a text, such as u1888888.

```
re.findall(r'u1\d\d\d\d\d\d\d')
```

**\s** | Matches whitespace characters, which include the \t (tab space), \n (new line), \r (return), and space characters.

**\S** | Matches non-whitespace characters.

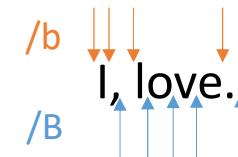
```
print(re.findall(r"\S","I love python3.")) # ['I', 'l', 'o', 'v', 'e', 'p', 'y', 't', 'h', 'o', 'n', '3', '.']
```

**\b** | matches the **empty string** (zero-width character, not blank space) at the beginning or end, i.e. boundary of a word (\w), in other words, between \w and \W.

**\B** | matches the any position that is not a word boundary \b.

```
print(re.findall(r"\w\b","I, love.")) #['I', 'e']
```

```
print(re.findall(r"\w\B","I, love.")) # ['I', 'o', 'v']
```



# Special Characters

^ | matches the starting position of the string.

```
print(re.findall(r'^\w','I, love, python')) # ['I']
```

\$ | matches the ending position of the string.

```
print(re.findall(r'\w$','I, love, python')) # ['n']
```

- . | matches any character except line terminators like \n.
- \ | Escapes special characters or denotes character classes.
- A | B | Matches expression A or B.

# Exercise