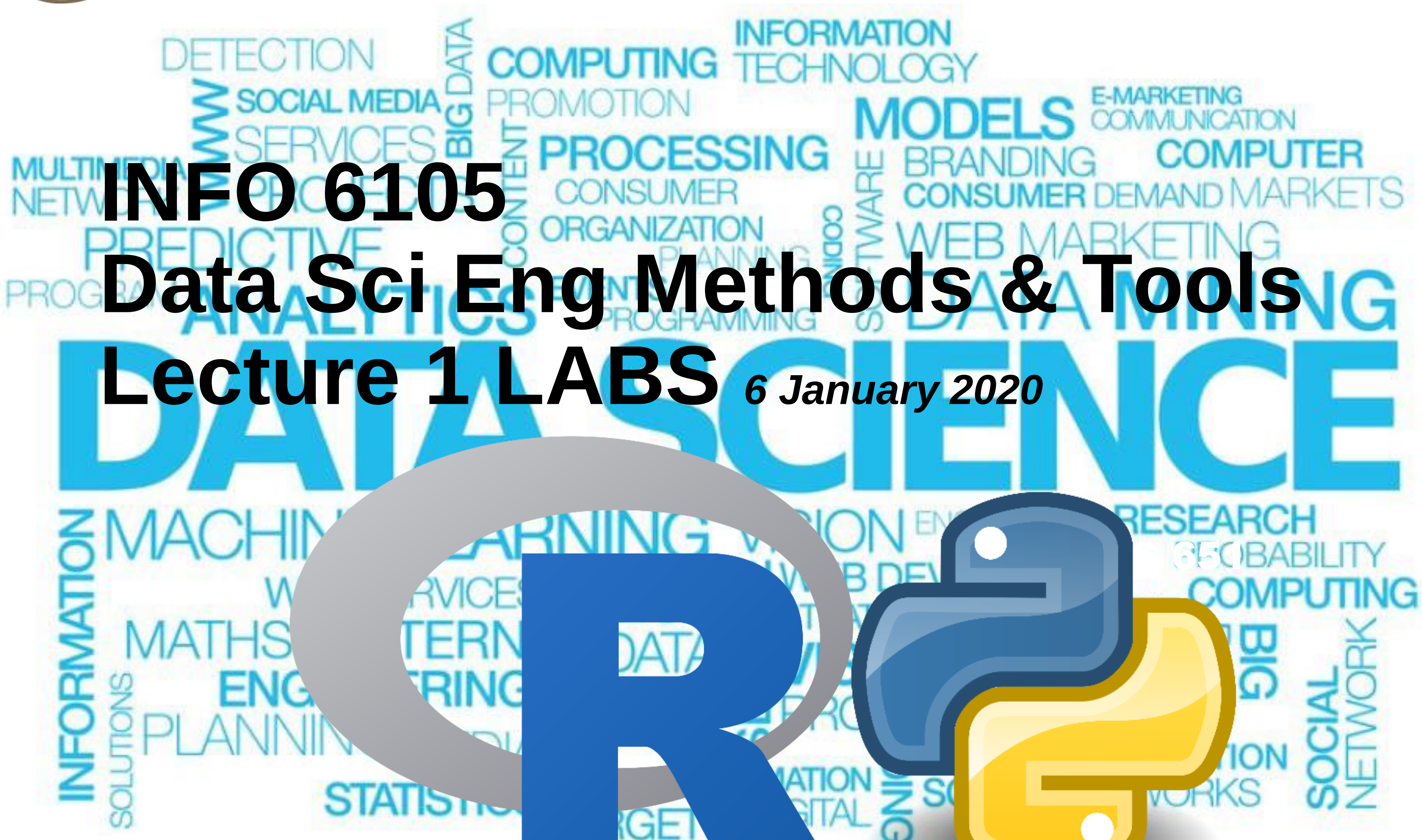
Northeastern University



INFO 6105 Data Sci Eng Methods & Tools Lecture 1 LABS

6 January 2020

Part 1
R

R vs Python

- Ross Ihaka and Robert Gentleman created the open-source language R in 1995 as an implementation of the S programming language
  - Purpose was to develop a language that focused on delivering a better and more user-friendly way to do data analysis, statistics and graphical models
  - CRAN, a huge repository of curated R packages to which users can easily contribute
    - https://cran.r-project.org/

Preliminaries

- Install *R*, *RStudio*, and the *dplyr* and *ggplot2* packages
  - In Rstudio console:
    ```
    install.packages("dplyr")
    install.packages("ggplot2")
    ```
- Set the working directory to the directory of the RHandsOn zip file on blackboard, as follows:
  - Session -> Set Working Directory -> Choose Directory...
  - Navigate and Open the /programs directory
- *Focus console after executing from source* option: Moves the focus to the console after executing a line or selection of code within the source editor (you *will* thank me)
  - Tools -> Global Options -> Code Editing...
  - Check "Focus console after executing from Source"

Topics

1. Data manipulation, including package dplyr
2. Graphics, including package ggplot2
3. Basic statistical models: linear and logistic regression

0-intro.R

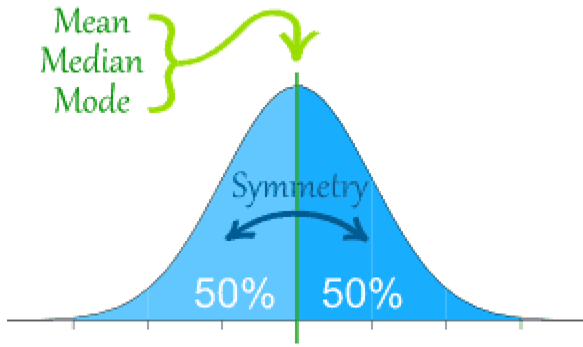
- Open the file 0-intro.R [File - Open file...]
- If the file appears empty, then..
  - `File | Reopen with Encoding | UTF-8`
- Read and execute instructions one by one to familiarize yourself with R commands..

Normal distribution (rnorm)

- **We say the data is "normally distributed“ when:**
  - **The probability density of the normal distribution is:**

$$f(x,\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

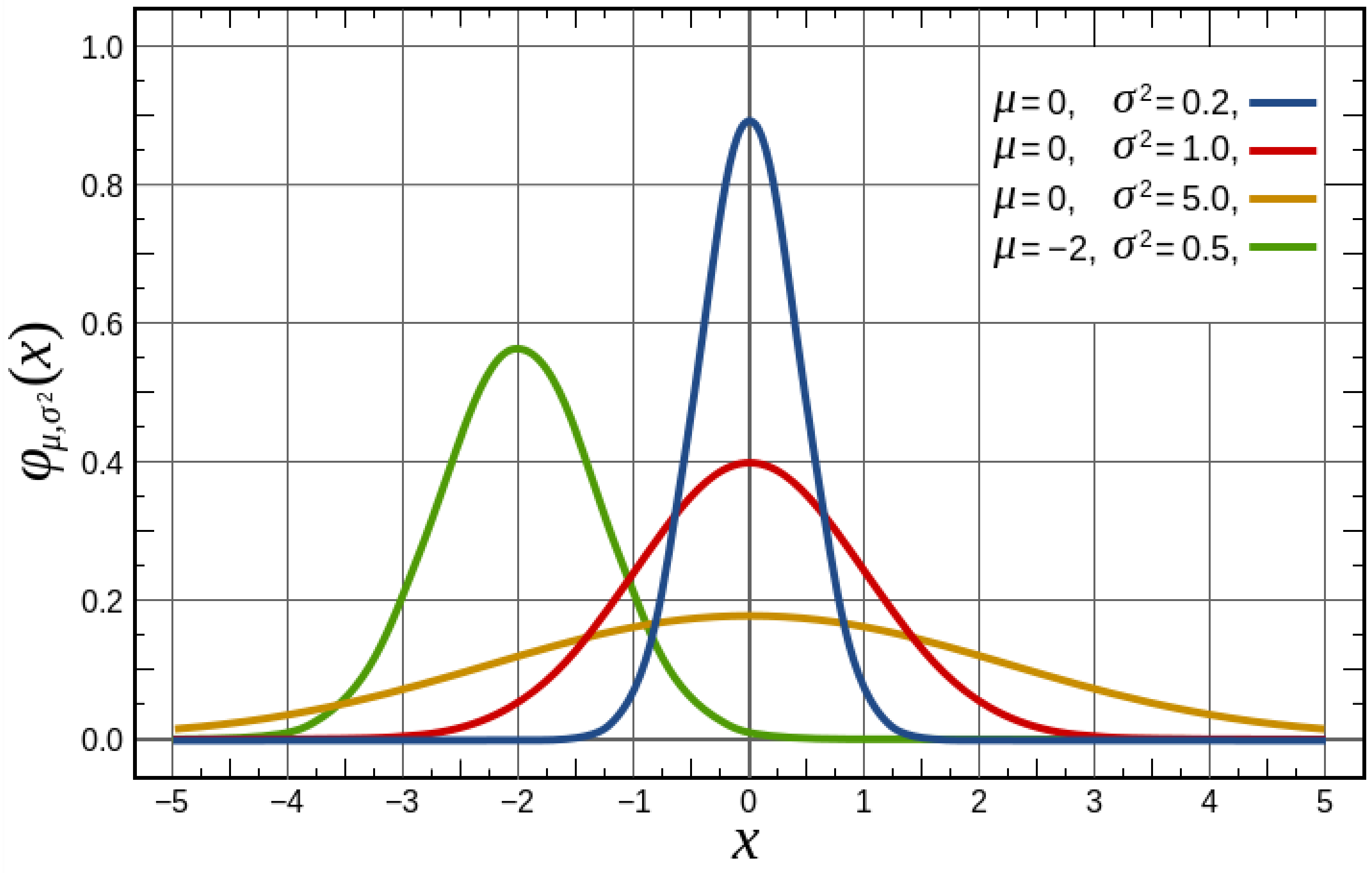
  - **Here, μ is the mean or expectation of the distribution (and also its median and mode). The parameter σ is its standard deviation; its variance is then σ^2**
  - **If μ = 0 and σ = 1 the distribution is called the *standard normal distribution* or the *unit normal distribution***



symmetry about the center

50% of values less than the mean and 50% greater than the mean

Normal Distributions



1-data.R

- **Open the file 1-data.R [File - Open file...]**
- ***CTRL-L to clear the Rstudio console***
- **college <- read.csv("../data/College.csv")**
  - **College.csv is a dataset of 777 different universities and colleges in the US, and contains information stored in a number of variables**
- **What do you think would happen if you try to load a file that is bigger than the amount of RAM on your laptop?**

dplyr

- Solves most common data manipulation operations, so that your options are helpfully constrained when thinking about how to tackle a problem (replaces plyr)
- Does split-apply-combine (SAC) logic
  - Best analogy is the GROUP BY statement in SQL
  - http://www.r-bloggers.com/do-your-data-janitor-work-like-a-boss-with-dplyr/
- Provides simple functions that correspond to the most common data manipulation verbs, so that you can easily translate your thoughts into code
- Uses efficient data storage back-ends, so that you spend as little time waiting for the computer as possible

dplyr (more)

- Abstracts away how your data is stored, so that you can work with data frames, data tables and remote databases using the same functions
  - This lets you think about what you want to achieve, not the logistics of data storage
- Provides a thoughtful default `print()` method so you don't accidentally print pages of data to the screen
- Compared to base functions, dplyr is more consistent: Functions have the same interface so that once you've mastered one, you can easily pick up the others

dplyr:filter()

- filter() allows you to select a subset of the rows of a data frame
  - The first argument is the name of the data frame, and the second and subsequent are filtering expressions evaluated in the context of that data frame
  - filter(flights, month == 1, day == 1) is equivalent to:
  - flights[flights$month == 1 & flights$day == 1,]

dplyr:select()

- Allows you to rapidly zoom in on a useful subset using operations that usually only work on numeric variable positions
  - # Select columns by name
    select(flights, year, month, day)

dplyr: summarize()

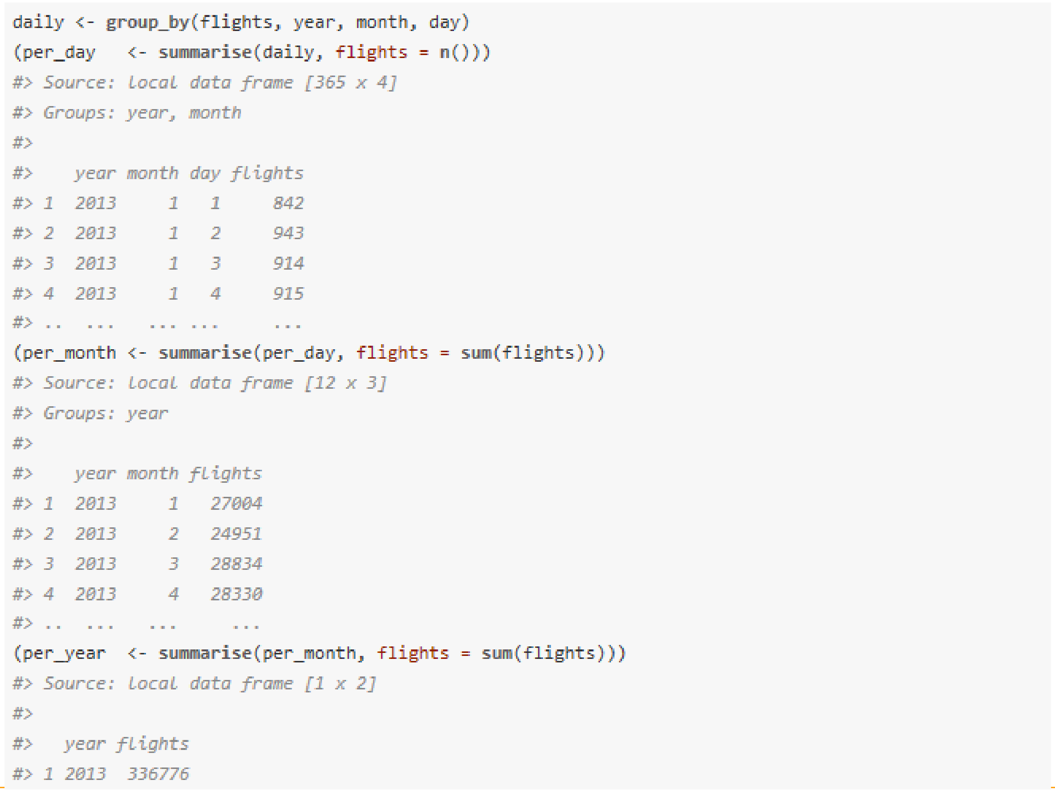
- You use summarize() with aggregate functions, which take a vector of values, and return a single number
  - There are many useful functions in base R like *min(), max(), mean(), sum(), sd(), median(), n()* (number of observations in the current group), *n_distinct(x)* (count the number of unique values
in x)

Grouped Ops

- select() now begins to look a lot like SQL’s SELECT operator, and group_by() like SQL’s GROUP BY operator..

Rolling up datasets

- **When you group by multiple variables, each summary peels off one level of the grouping**
  - **That makes it easy to progressively roll-up a dataset:**

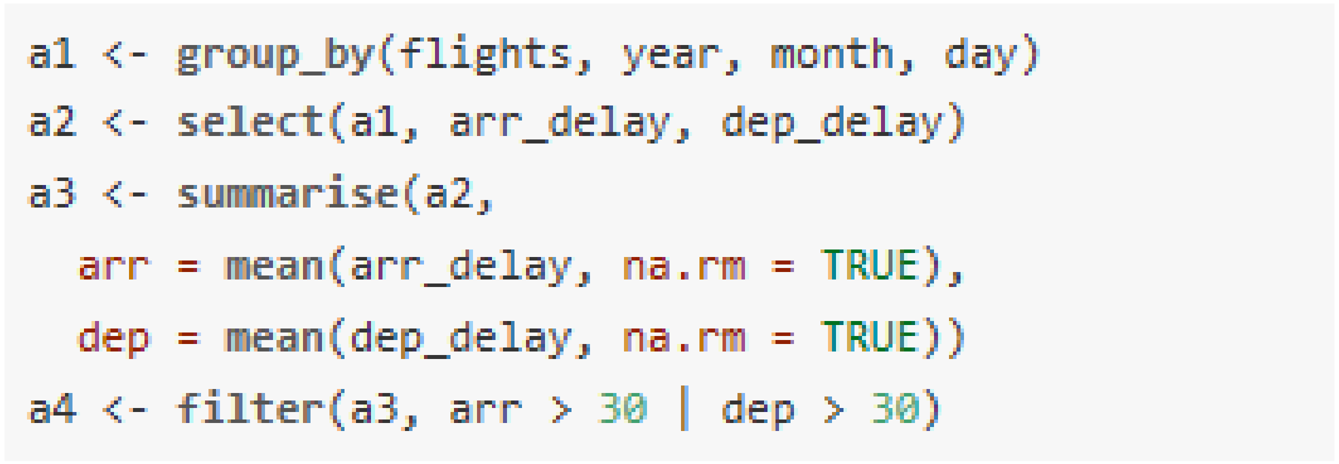


```
daily <- group_by(flights, year, month, day)
(per_day   <- summarise(daily, flights = n()))
#> Source: local data frame [365 x 4]
#> Groups: year, month
#>
#>    year month day flights
#> 1  2013     1   1     842
#> 2  2013     1   2     943
#> 3  2013     1   3     914
#> 4  2013     1   4     915
#> ..  ...   ... ...     ...
(per_month <- summarise(per_day, flights = sum(flights)))
#> Source: local data frame [12 x 3]
#> Groups: year
#>
#>    year month flights
#> 1  2013     1   27004
#> 2  2013     2   24951
#> 3  2013     3   28834
#> 4  2013     4   28330
#> ..  ...   ...     ...
(per_year  <- summarise(per_month, flights = sum(flights)))
#> Source: local data frame [1 x 2]
#>
#>   year flights
#> 1 2013  336776
```

Chaining

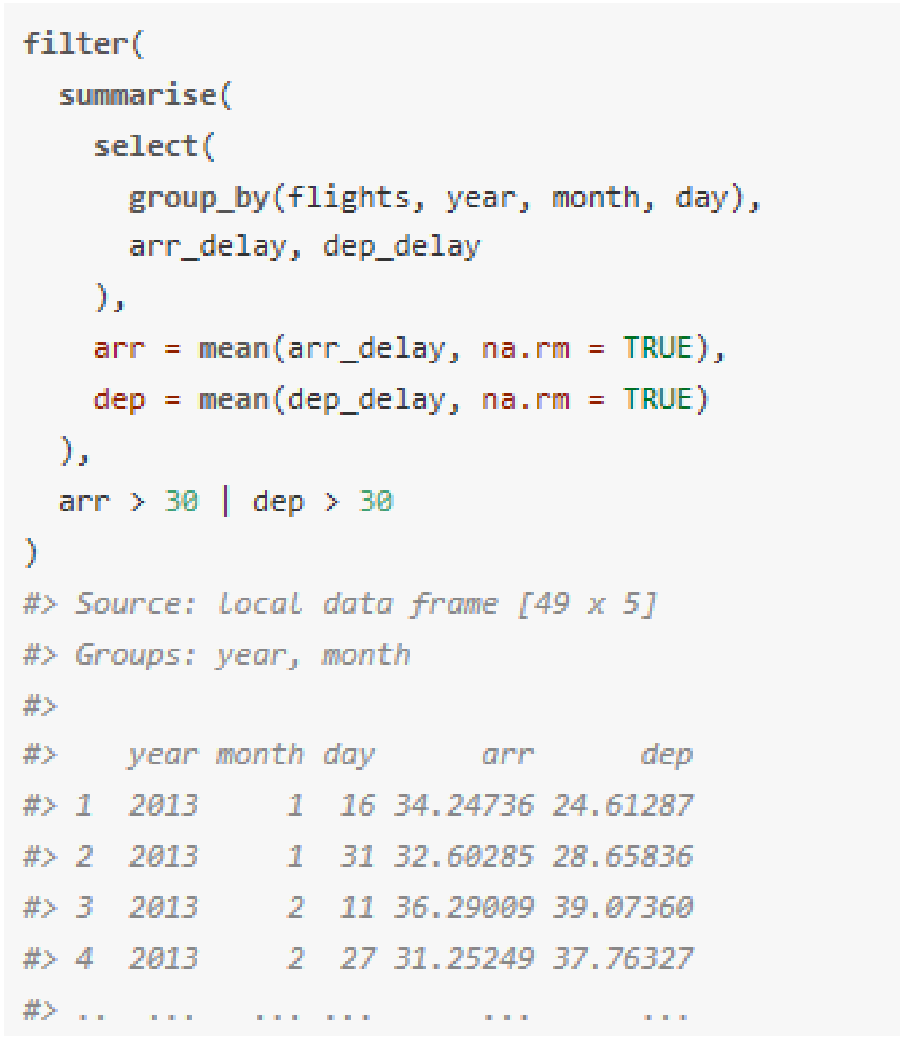
- The dplyr API is ***functional*** in the sense that function calls don't have side-effects
- If you want to do many operations at once, you either have to do it step-by-step, or
- If you don't want to save the intermediate results, you need to wrap the function calls inside each other

Step by step



```
a1 <- group_by(flights, year, month, day)
a2 <- select(a1, arr_delay, dep_delay)
a3 <- summarise(a2,
  arr = mean(arr_delay, na.rm = TRUE),
  dep = mean(dep_delay, na.rm = TRUE))
a4 <- filter(a3, arr > 30 | dep > 30)
```

Wrapping function calls



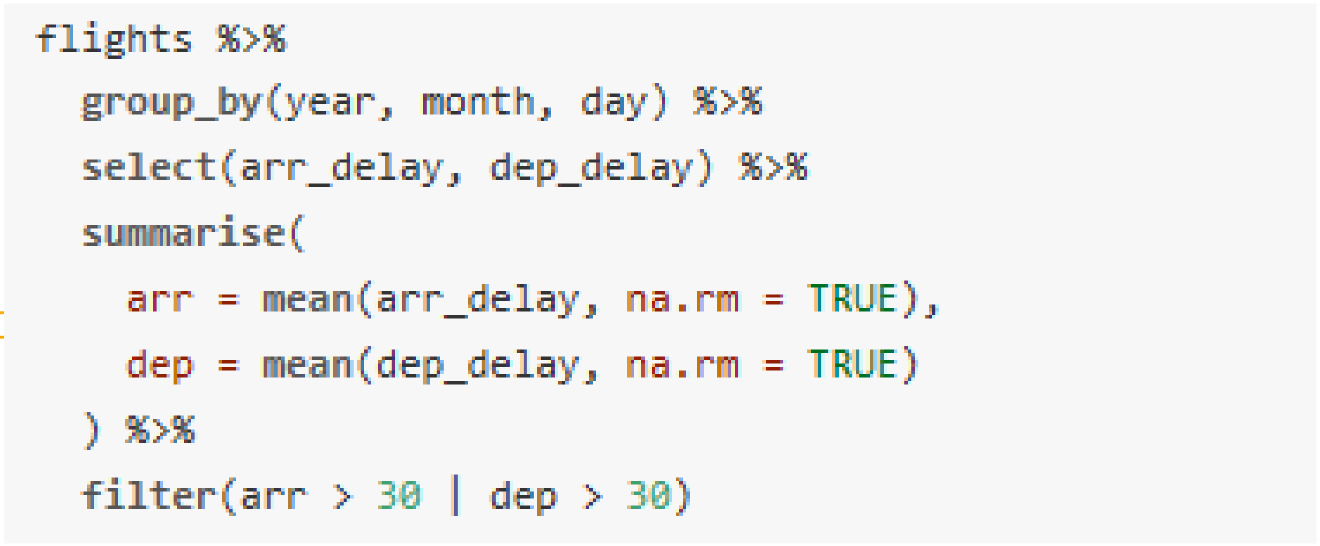
```
filter(
  summarise(
    select(
      group_by(flights, year, month, day),
      arr_delay, dep_delay
    ),
    arr = mean(arr_delay, na.rm = TRUE),
    dep = mean(dep_delay, na.rm = TRUE)
  ),
  arr > 30 | dep > 30
)
#> Source: local data frame [49 x 5]
#> Groups: year, month
#>
#>    year month day      arr      dep
#> 1  2013     1  16 34.24736 24.61287
#> 2  2013     1  31 32.60285 28.65836
#> 3  2013     2  11 36.29009 39.07360
#> 4  2013     2  27 31.25249 37.76327
#> ..  ...   ... ...      ...      ...
```

- **Difficult to read because the order of the operations is from inside to out, and the arguments are a long way away from the function..**

The pipe (%>%) operator

Using %>%

- `x %>% f(y)` turns into `f(x, y)`
- Use it to rewrite multiple operations so you can read from left-to-right, top-to-bottom:



```r
flights %>%
  group_by(year, month, day) %>%
  select(arr_delay, dep_delay) %>%
  summarise(
    arr = mean(arr_delay, na.rm = TRUE),
    dep = mean(dep_delay, na.rm = TRUE)
  ) %>%
  filter(arr > 30 | dep > 30)
```

dplyr:mutate()

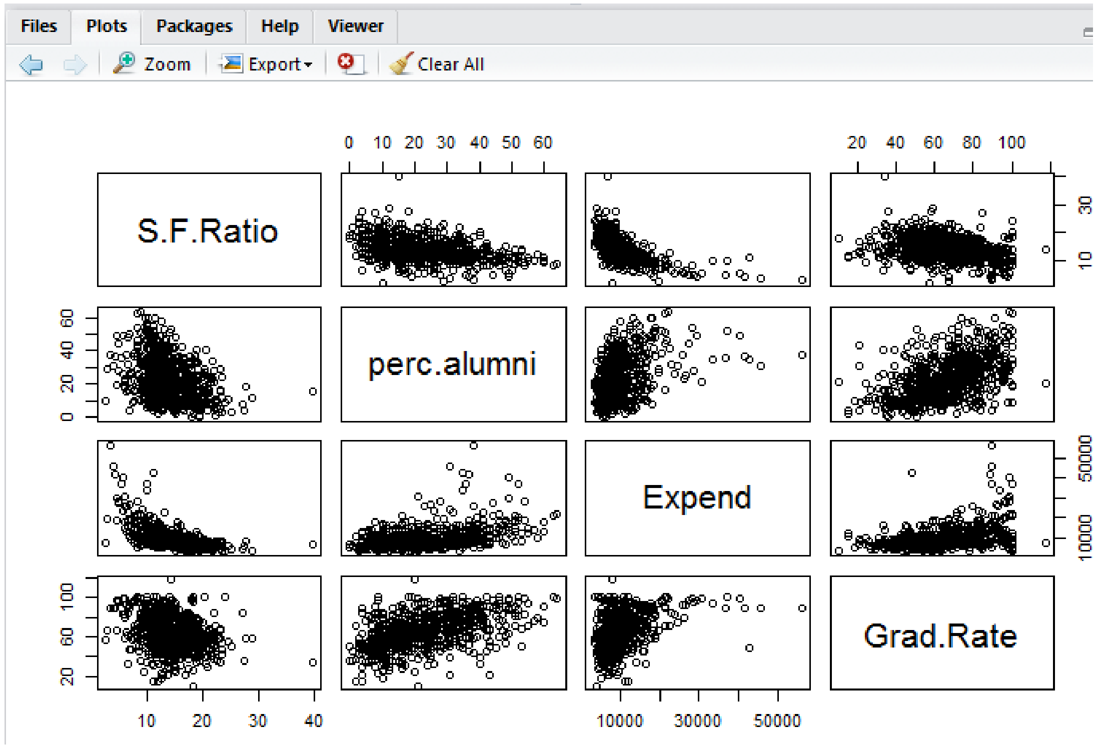
- Adds new columns that are functions of existing columns
  - mutate(flights,
    gain = arr_delay - dep_delay,
    speed = distance / air_time * 60)

2-graphics.R

- **Open the file 2-graphics.R [File - Open file...]**
- ***CTRL-L to clear the Rstudio console***

Base R

- Plot() is pretty decent!
  - pairs(college[,c(16:19)]):



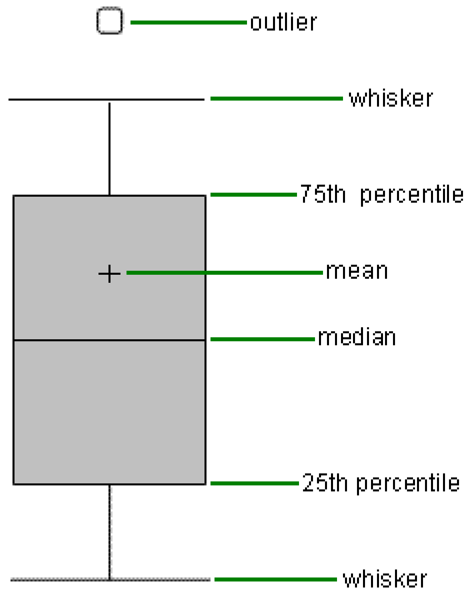
ggplot2

- Designed to work in a layered fashion
  - Example:

```
## aes = aesthetics
p <- ggplot(college, aes(x=S.F.Ratio, y=Grad.Rate))
p + geom_point()
p + geom_point(aes(colour = Private))
```

Box plot

- The box plot (a.k.a. box and whisker diagram) is a standardized way of displaying the distribution of data based on the five number summary: ***minimum, first quartile, median,***
third quartile, and
maximum



Facets

- Faceting approach supported by ggplot2 partitions a plot into a matrix of panels
  - Each panel shows a different subset of the data
  - facet_wrap(~cell) - univariate: create a 1-d strip of panels, based on one factor, and wrap the strip into a 2D matrix
  - facet_grid(row~col) - (usually) bivariate: create a 2D matrix of panels, based on two factors
- Good example of faceting data analysis:
  - http://sape.inf.usi.ch/quick-reference/ggplot2/facet

3-stats.R

- **Open the file 3-stats.R [File - Open file...]**
- ***CTRL-L to clear the Rstudio console***

What for

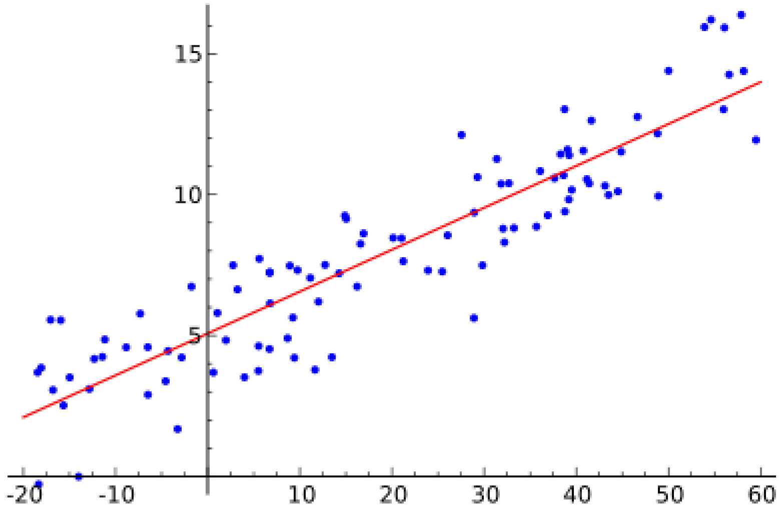
- We use statistical analysis for:
  - *Inference* - making conclusions based on data
  - *Prediction* - what will happen when I observe new data?
  - And we create *models* to do both of those things
- *"All models are wrong - some are useful“* - George E. P. Box

It’s been said that..

- Our brain simply consists of a bunch of *predictors*, based on *models* we build for ourselves in our lifetimes..

Linear Regression

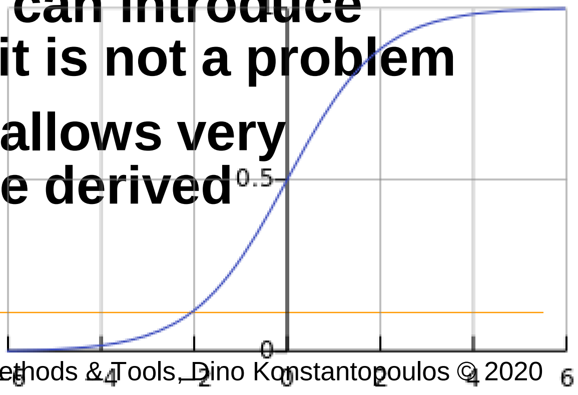
- Regression is an approach for modeling the relationship between a scalar *dependent* variable y and one or more explanatory variable (*independent* variable) x
  – In linear regression, data are modeled using linear predictor functions
  – Linear regression models are often fitted using the least squares approach, but they may also be fitted in other ways..



$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon},$$

Logistic Regression

- Logistic regression describes a kind of classification model in which predictor variables are combined with linear weights and then passed through a soft-limit function that limits the output to the range [0, 1]
- Logistic regression is closely related to other models such as:
  - Perceptron (where the soft limit is replaced by a hard limit)
  - *Neural networks* (where multiple layers of linear combination and soft limiting are used)
  - *Naive Bayes* (where linear weights are determined strictly by feature frequencies assuming independence)
- Logistic regression can't separate all possible classes, but in very high dimensional problems or where you can introduce new variables by combining other predictors, it is not a problem
- Mathematical simplicity of logistic regression allows very efficient and effective learning algorithms to be derived



7-oop.R

- Open the file 7-oop.R [File - Open file...]
- *CTRL-L to clear the Rstudio console*
- Although the latest built-in OOP model, RC ("reference classes"), is superior in many ways to S4 and its immediate predecessor model S3, both S3 and S4 are still widely used
  - Quite a few common R language functions were created using the S4 model
  - The RC model is much more like the C# OOP model you;re used to, but most R programmers come from a strictly R programming background and feel more comfortable with S3 or S4
  - One of the significant improvements in the S4 OOP model compared to the S3 model is that you can specify types for the fields. In addition to the atomic types "integer," "character," and "numeric" used in the demo, you can specify composite types such as "vector," "matrix," "array," and "data.frame"

Class Methods

- An S4 class encapsulates data fields inside a `setClass` function, but doesn't encapsulate class methods
- Instead, S4 class methods are defined by pairs of special R functions named `setMethod` and `setGeneric`
- There's a special initialize function that's defined only by `setMethod` but not with `setGeneric`
  - The special initialize function corresponds to a C++ constructor

Equivalents

- C#

```
Person p1 = new Person; // Default values for fields
p1.empID = 65565;
p1.lastName = "Adams"; // Change name
p1.hireDate = DateTime.Parse("2010/09/15");
p1.payRate = 43.21;
p1.Display;
int tenure = p1.YearsService;
```

- R

```
p1 <- new("Person") # default values for fields
p1@empID <- as.integer(65565)
p1@lastName <- "Adams"  # change name
p1@hireDate <- "2010-09-15"
p1@payRate <- 43.21
display(p1)
tenure <- yearsService(p1)
```

Resources

- Quick-R
  http://www.statmethods.net
- *An Introduction to Statistical Learning with Applications in R*, by James et al (ISLR)
- *Data Wrangling* handout

Part 2

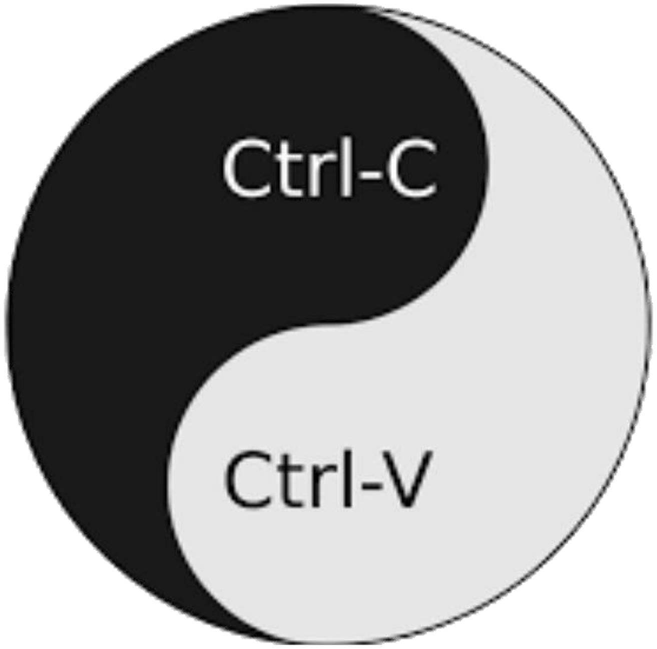
PYTHON QUICK INTRODUCTION

Python 3.x

- Python is a *managed* language, with its own runtime engine, just like Java and C#
  - Surprised?
- On the Mac, it comes installed with, but it's the *wrong* version for this class
  - You haver python 2.7, but we run python 3.x in class
  - On the Mac, you might have to run python3 instead of python on the command line
  - Make sure to install Anaconda with Python 3.7 (or 3.8 if available)!
- On Windows, you need to install a Python runtime
  - Make sure to install the 3.8 version, not the 2.7 one
  - Make sure to install it on the coot of C:\ (not in Program Files)
- Make sure to install 64 bit version of Anaconda, with Python 3.x
  - Run this in a cell: `!python --version`

Python basics

- Copy paste the code in the following slides in a new Anaconda Python notebook
- Google the concepts



Container types

- Tuple: immutable sequence
  - () (23,) (23, 45) tuple('ciao’)
- List: mutable sequence (a "vector")
  - [] [23] [23, 45] list('ciao’)
- Set: mutable:
  - set() set((23,)) set('ciao')
  - immutable variant (frozenset)
- Dict: map key→value by hashtable
  - {} {2:3} {4:5, 6:7} dict(ci='ao')
- All containers support:
  - `len(c)`, looping (`for x in c`), membership testing (`if x in c`)

Lists

- Most fundamental data structure in Python: Ordered collection
  - `a = [1,2,3]`
  - `b = [4.5.6]`
  - `List_of_lists = [a, b, []]`
  - `s = sum(a)`
  - `one = a[0]`
  - `three = a[-1]  #last element`
  - `two = a[-2]  #next-to-last eleme`
  - `digits = range(10)`
  - `first_three = digits[:3]`
  - `minus_first_three = digits[3:]`
  - `one_to_four = digits[1:5]`
  - `copy_of_digits = digits[:]`
  - `x.extend([10,11,12])`
  - `x.append(13)`
  - `x, y = [1,2]`

Sorting

- `x = [4,1,2,3]`
- `y = sorted(x)   #x remains unchanged`
- `x.sort()   #x is sorted in place`

-
  ```
  wc = sorted(word_counts.items(),
              key = lambda (word, count): count
              reverse = True) #instead of comparing
              elements themselves, compare results
              of a function that you specify with
              'key'
  ```
- Built-in `bisect` module implements binary search and insertion into a sorted list
  - `Bisect.bisect` finds the location where an element should be inserted to keep the list sorted

  -
    ```
    import bisect
    c = [1,2,2,2,3,4,7]
    bisect.bisect(c,2)   #4
    bisect.bisect(c,5)   #6
    bisect.insort(c, 6) #[1,2,2,2,3,4,6,7]
    ```

Tuples

- Lists' immutable cousins
  - `mylist = [1, 2,3]`
  - `mytuple = (1,2,3)`
- Tuples are a convenient way to return multiple values from functions:

```
def sum_product(x, y):
    return (x + y), (x * y)
s, p = sum_product(10, 10)
```

- Multiple assignments:
  - `x, y = 1, 2`
- Python variable swap:
  - `x, y = y, x`

Dictionaries

- Lists of key/value pairs, or *named arrays*
  - `empty = {}`
  - `also_empty = dict()`
  - `grades = {“dino”: 3.9, “elon”: 4.0}`
  - `elon_grade = grades[“elon”]`
  - `elon_grade = grades.get(“elon”, 0)`
  - `dinograde_p = “dino” in grades`

  -
    ```
    json = {
        “title”: “my blog”,
         “hashtags”: [“#bigdata”, “#crypto”, “#quantum”]
    }
    ```
  - `json.keys()`
  - `json.values()`
  - `json.items()`
- Dictionary keys are immutable
  -

Defaultdict

- Like a regular dictionary, except when you try to look up a key that isn't there, it first adds a value for it using a zero-argument function you provide when you create it
- Useful whem using dictionaries to collect results by some key and don't want to check repeatedly for key existence

  -
    ```
    From collections import defaultdict
    word_counts = defaultdict(int)
    for word in document:
      word_counts[word] += 1
    ```

  -
    ```
    dd = defaultdict(dict)
    dd["dino"]["City"] = "Boston"
                    #{"dino": {"City": "Boston"}}
    ```

Counter

-
  ```
  From collections import Counter
  word_count = Counter([“to”, “be”, “or”, “not”,\
  “to”, “be”])
  ```

Sets

- Unordered collection of distinct elements
  - `s = set()`
  - `s.add(1)`
  - `s.add(2)`
  - `s.add(2)`
  - `p = 2 in s`
- Performance:

  -
    ```
    stopwords_list = [“a”, “the”, …
    p = “hello” in stopwords  #slow
    ```

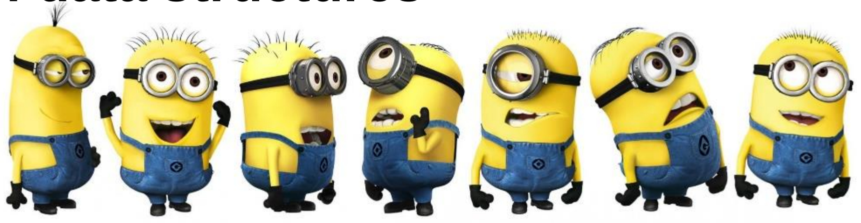
  -
    ```
    stopwords_set = set(stopwords_list)
    p = “hello” in stopw-rds_set  #fast
    ```
- Distinct:
  - `word_list = [‘the’, “cat”, “jumps”, …]`
  - `distinct_word_list = set(world_list)`

List comprehensions

- Transformations of lists:
  - `even_numbers = [x for x in range(100) if x % 2 == 0 ]`
  - `even_set = {x for x in range(100) if x % 2 == 0 }`
  - `zeroes = [ 0 for _ in range(100)]`

  -
    ```
    pairs = [ (x,y)
                for x in range(100)
                for y in range(100) ]  #10,000 pairs
    ```

  -
    ```
    some_tuples = [(1,2,3), (4,5,6), (7,8,9)]
    flattened = [x for tup in some_tuples for x in tup]
    flattened  #[1,2,3,4,5,6,7,8,9]
    ```
- List comprehensions is the pythonic equivalent of LINQ in .NET
- We'll use list comprehensions *a lot* in data science because they represent *anamorphisms* (unfolds or maps) and *catamorphisms* (projections) of data structures
  - Get ready for this!



Generators

- *Generators*, sometimes called *Coroutines*, are sequences you can iterate over, but which are only produced lazily (as needed)
- You can create generators with functions and yield:

  -
    ```
    def lazy_range(n):
      """a lazy version of range()"""
      i = 0;
      while i < n:
        yield i
        i += 1
    ```

  -
    ```
    # to consume yielded values:
    for i in lazy_range(10)
      print(i)
    ```
- You may also create generators by using list comprehensions wrapped in parenses:
  - `lazy_ints_under_100 = (i for i in range(100))`

Iterators

- Standard `itertools` library has a collection of generators for common data algorithms

  -
    ```
    import itertools
    first_letter = lambda x: x[0]
    names = [ 'Alex', 'Aria', 'Wally', 'Will',
    'Ariana', 'Steve']
    for letter, names in itertools.groupby(names,
    first_letter):
      print(letter, list(names))
     # A ['Alex, 'Aria']
       W ['Wally', 'Will']
       A ['Ariana']
       S ['Steve']
    ```

Control flow

-
  ```
  if 1 == 2:
    print(“uh-oh”)
  elif 1 == 3:
    print(“uh-oh-again”)
  else:
    print(“whew..”)
  ```

-
  ```
  while x < 100:
    print(x)
  ```

-
  ```
  for x in range(100):
    if x < 100:
      continue
    if x > 100:
      break;
    print(x)
  ```

Enumerations

```
#nicely functional
for i, document in enumerate(documents):
    do_something(i, document)
```

```
#unpythonic
for i in range(len(documents)):
    document = documents[i]
    do_something(I, document)
```

```
#also unpythonic
i = 0
for document in documents:
    do_something(i, document)
    i += 1
```

File IO

-
  ```
  path = ‘myfolder/mybigdata.txt’
  ```

-
  ```
  f = open(path)
  for line in f:
    print(line)
  #EOL marker intact
  ```

-
  ```
  Lines = [x.rstrip() for x in open(path)]
  #EOL-free
  ```
- Using analog:

  -
    ```
    with open(path) as f:
      lines = [x.rstrip() for x in f]
    #automatically closes the file when exiting the
    with block
    ```

-
  ```
  with open(path, ‘rb’) as f:
    data.decode(‘utf8’)
  ```

Object Oriented Python

```
 Class Set:
    def __init__(self, values=None):
      “””ctor”””
      self.dict = {} #each instance has its own
                     #dict which is what we use
                     #to track membership
      if values is not None:
        for value in values:
          self.add(value)

    def add(self, value):
      self.dict[value] = True

    def contains(self, value):
      return value in self.dict

    def remove(self, value):
      del self.dict[value]
```

Currying

- **Partially applying functions to create new functions**
  - def exp(base, power):
    return base ** power
  - def two_to_the(power):
    return exp(2, power)
  - From functools import partial
    two_to_the = partial(exp, 2)
  - Print(two_to_the(3))

Function Oriented puzzle

- **Let's say we want to create a higher-order function that takes as input some function `f` and returns a new function that for any input returns twice the value of `f`**

```
def dpubler(f):
  def g(x):
    return 2 * f(x)
  return g
```

- **Works in most cases:**

```
def f_plus_1(x)::
  return x + 1;
g = doubler(f_plus_1)
print(g(3))  # 8 = (3 + 1) * 2
```

- **But:**

```
def sum(x, y)::
  return x + y;
g = doubler(sum)
print(g(1,2))
```

`args` and `kwargs`

- What we need is a way to specify a function that takes arbitrary arguments:

```
def magic(*args, **kwargs):
  print (“unnamed args: “, args)
  print (“keyword args: “, kwargs)
magic(1,2, key1 = “nu”, key2 = ’rocks!”);
```

- `args` is a tuple of its unnamed arguments and `kwargs` is a dictionary of its named arguments. So now we can:

```
def dpublerr(f):
  “””works no matter the inputs”””
  def g(*args, **kwargs):
    “””pass all arguments to f”””
    return 2 * f(*args, **kwargs)
  return g
```

- And now:

```
g = doublerr(sum)
print g(1, 2)  # 6:
```

Zippers

- `zip` transforms multiple lists into a single list of tuples of corresponding elements
  - `list1 = [‘a’, ‘b’, ‘c’]`
  - `list2 = [1, 2, 3]`
  - `zipper = zip(list1, list2)  #[(‘a’, 1), (‘b’, 2), (‘c’, 3)]`
  - `Orig_letters, orig_numbers = zip(*zipper)`
    `# * performs argument unpacking`
  - `def add(a, b): return a + b`
  - `add(1, 2)   #3`
  - `add([1,2])  #TypeError!`
    `add(*[1,2]) #3`

Bonus Homework (to practice your Python)

- Complete missing parts of R labs and try to do equivalent functionality in Python using a Python notebook (.ipynb)
  - Perform regressions on college dataset
  - Deliver `.R` and `.ipynb` files to TAs
  - Ask TAs how to post homework on blackboard
- To practice:
  - Find python videos on youtube
  - The good ones are those that don’t put you to sleep after 10 minutes. If you’re still awake after 10 minutes and you feel like you’re learning, then

CLASS
DISMISSED
iStockphoto