COMP 5070

Statistical Programming *for* Data Science

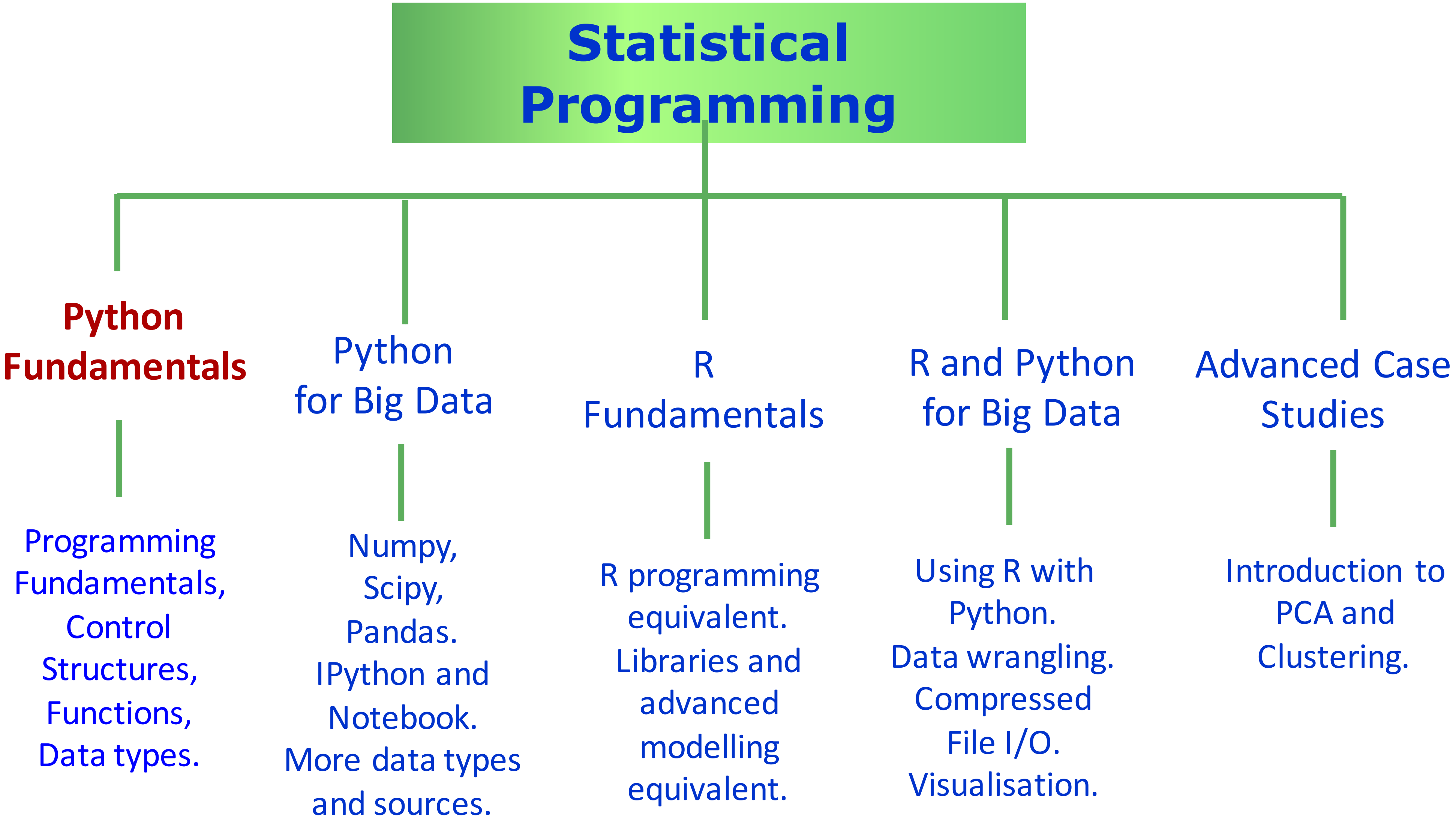
Weeks 1-3:

Python Fundamentals

```
Python 2.x.y (2004-08-15 16:23:42)
Type "help", "copyright", "credits"
>>>_
```

University of
South Australia

The course at a glance



Statistical Programming

Python Fundamentals

Programming Fundamentals, Control Structures, Functions, Data types.

Python for Big Data

Numpy, Scipy, Pandas. IPython and Notebook. More data types and sources.

R Fundamentals

R programming equivalent. Libraries and advanced modelling equivalent.

R and Python for Big Data

Using R with Python. Data wrangling. Compressed File I/O. Visualisation.

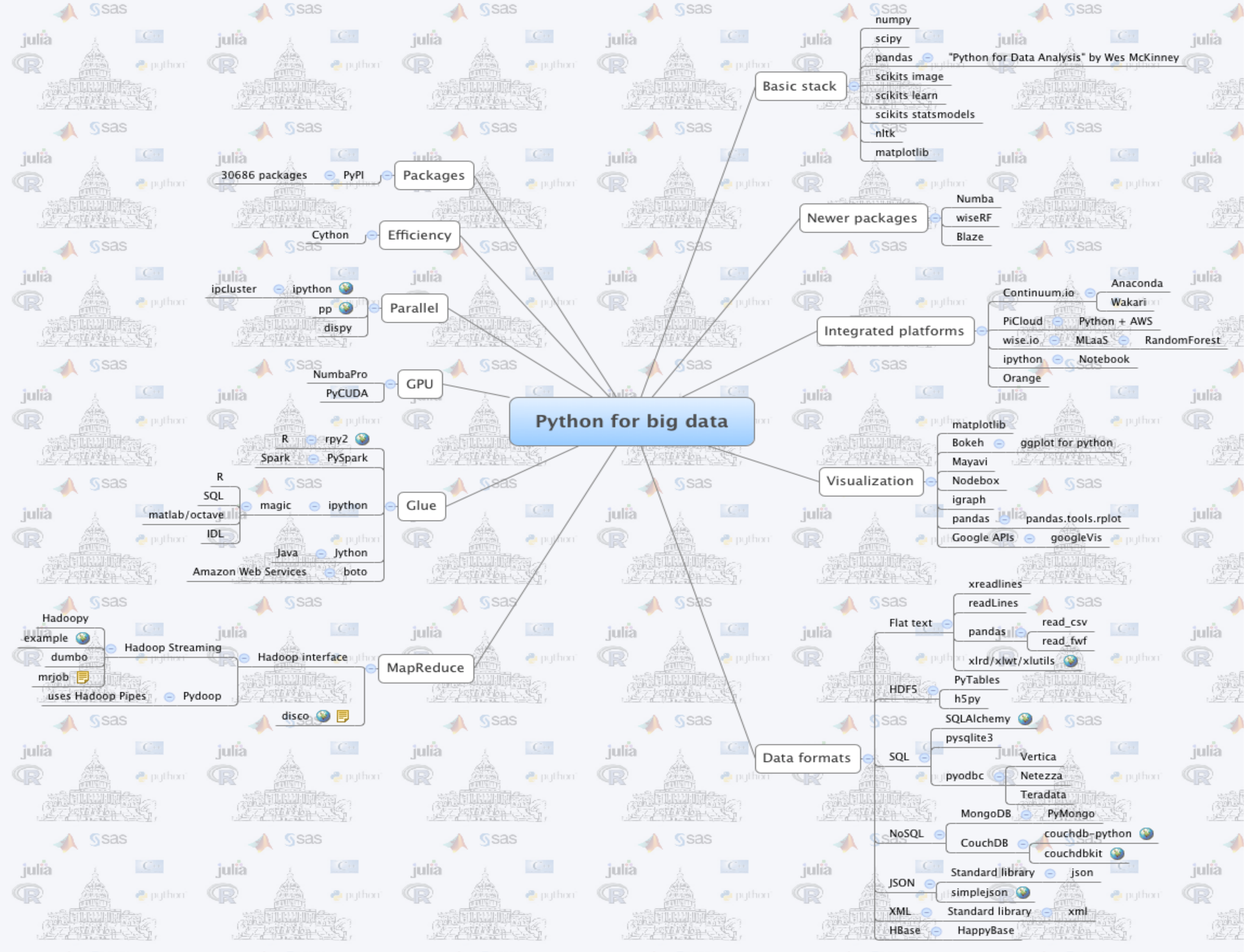
Advanced Case Studies

Introduction to PCA and Clustering.

First up …

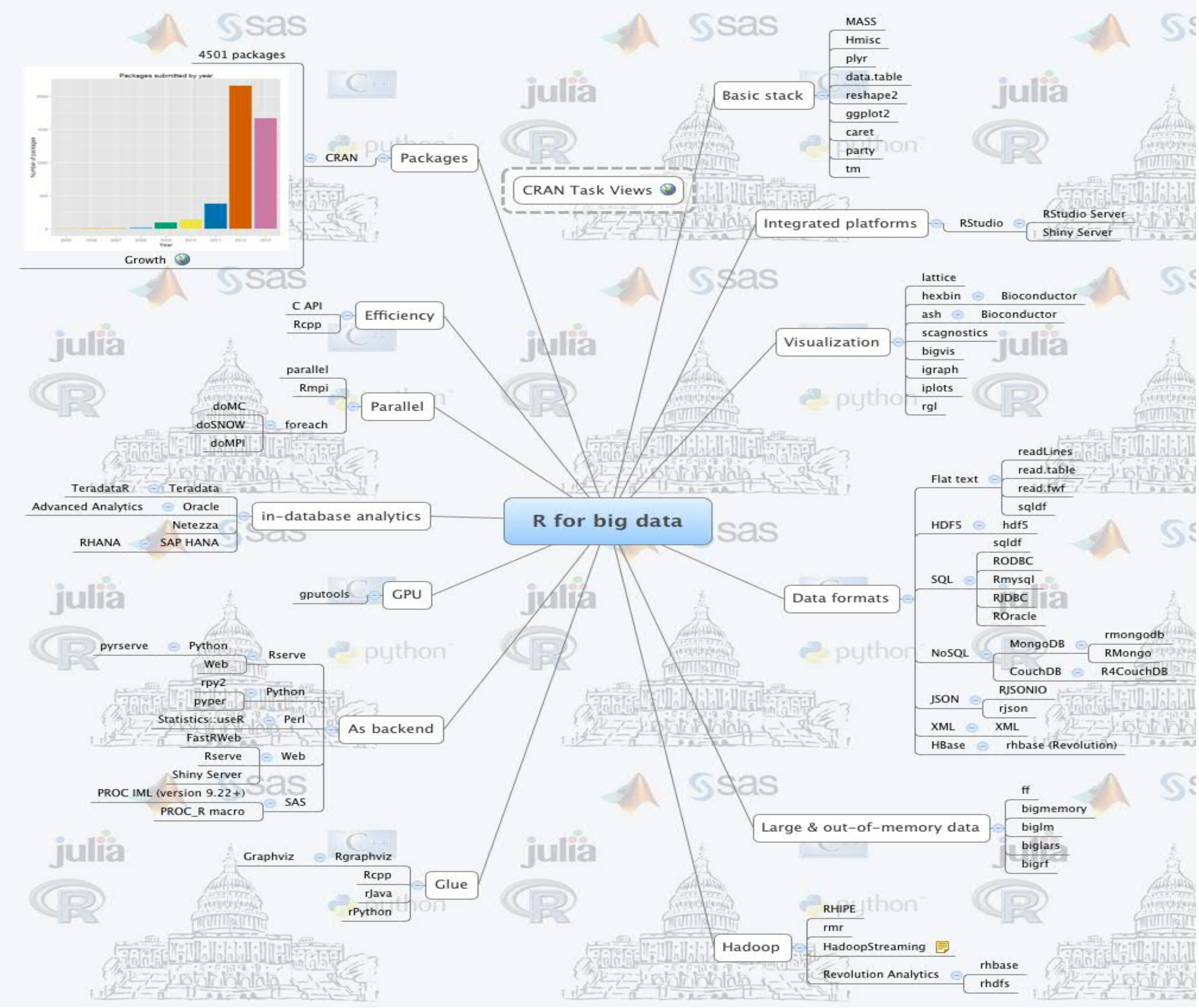
The Basics

- The Big Picture
- Introduction
  - □ What is a programming language?
  - □ Python
  - □ Fundamentals
  - □ Hello World!
- Data Types: an introduction
- Precedence and Associativity
- The `import` statement
- Python practice and exercises



Python for big data

- Basic stack
  - numpy
  - scipy
  - pandas
    - "Python for Data Analysis" by Wes McKinney
  - scikits image
  - scikits learn
  - scikits statsmodels
  - nltk
  - matplotlib
- Newer packages
  - Numba
  - wiseRF
  - Blaze
- Integrated platforms
  - Continuum.io
    - Anaconda
    - Wakari
  - PiCloud
    - Python + AWS
  - wise.io
    - MLaaS
      - RandomForest
  - ipython
    - Notebook
  - Orange
- Visualization
  - matplotlib
  - Bokeh
    - ggplot for python
  - Mayavi
  - Nodebox
  - igraph
  - pandas
    - pandas.tools.rplot
  - Google APIs
    - googleVis
- Data formats
  - Flat text
    - xreadlines
    - readLines
    - pandas
      - read_csv
      - read_fwf
    - xlrd/xlwt/xlutils
  - HDF5
    - PyTables
    - h5py
  - SQL
    - SQLAlchemy
    - pysqlite3
    - pyodbc
      - Vertica
      - Netezza
      - Teradata
  - NoSQL
    - MongoDB
      - PyMongo
    - CouchDB
      - couchdb-python
      - couchdbkit
  - JSON
    - Standard library
      - json
    - simplejson
  - XML
    - Standard library
      - xml
  - HBase
    - HappyBase
- Packages
  - PyPI
    - 30686 packages
- Efficiency
  - Cython
- Parallel
  - ipython
    - ipcluster
  - pp
  - dispy
- GPU
  - NumbaPro
  - PyCUDA
- Glue
  - rpy2
    - R
  - PySpark
    - Spark
  - ipython
    - magic
      - R
      - SQL
      - matlab/octave
      - IDL
  - Jython
    - Java
  - boto
    - Amazon Web Services
- MapReduce
  - Hadoop interface
    - Hadoop Streaming
      - Hadoopy
      - example
      - dumbo
      - mrjob
    - Pydoop
      - uses Hadoop Pipes
  - disco



- R for big data
  - Packages
    - CRAN
      - 4501 packages
      - Growth
  - CRAN Task Views
  - Basic stack
    - MASS
    - Hmisc
    - plyr
    - data.table
    - reshape2
    - ggplot2
    - caret
    - party
    - tm
  - Integrated platforms
    - RStudio
      - RStudio Server
      - Shiny Server
  - Efficiency
    - C API
    - Rcpp
  - Parallel
    - parallel
    - Rmpi
    - foreach
      - doMC
      - doSNOW
      - doMPI
  - in-database analytics
    - Teradata
      - TeradataR
    - Oracle
      - Advanced Analytics
    - Netezza
    - SAP HANA
      - RHANA
  - GPU
    - gputools
  - As backend
    - Rserve
      - Python
        - pyrserve
      - Web
    - Python
      - rpy2
      - pyper
    - Perl
      - Statistics::useR
    - Web
      - FastRWeb
      - Rserve
      - Shiny Server
    - SAS
      - PROC IML (version 9.22+)
      - PROC_R macro
  - Glue
    - Rgraphviz
      - Graphviz
    - Rcpp
    - rJava
    - rPython
  - Visualization
    - lattice
    - hexbin
      - Bioconductor
    - ash
      - Bioconductor
    - scagnostics
    - bigvis
    - igraph
    - iplots
    - rgl
  - Data formats
    - Flat text
      - readLines
      - read.table
      - read.fwf
      - sqldf
    - HDF5
      - hdf5
    - SQL
      - sqldf
      - RODBC
      - Rmysql
      - RJDBC
      - ROracle
    - NoSQL
      - MongoDB
        - rmongodb
        - RMongo
      - CouchDB
        - R4CouchDB
    - JSON
      - RJSONIO
      - rjson
    - XML
      - XML
    - HBase
      - rhbase (Revolution)
  - Large & out-of-memory data
    - ff
    - bigmemory
    - biglm
    - biglars
    - bigrf
  - Hadoop
    - RHIPE
    - rmr
    - HadoopStreaming
    - Revolution Analytics
      - rhbase
      - rhdfs

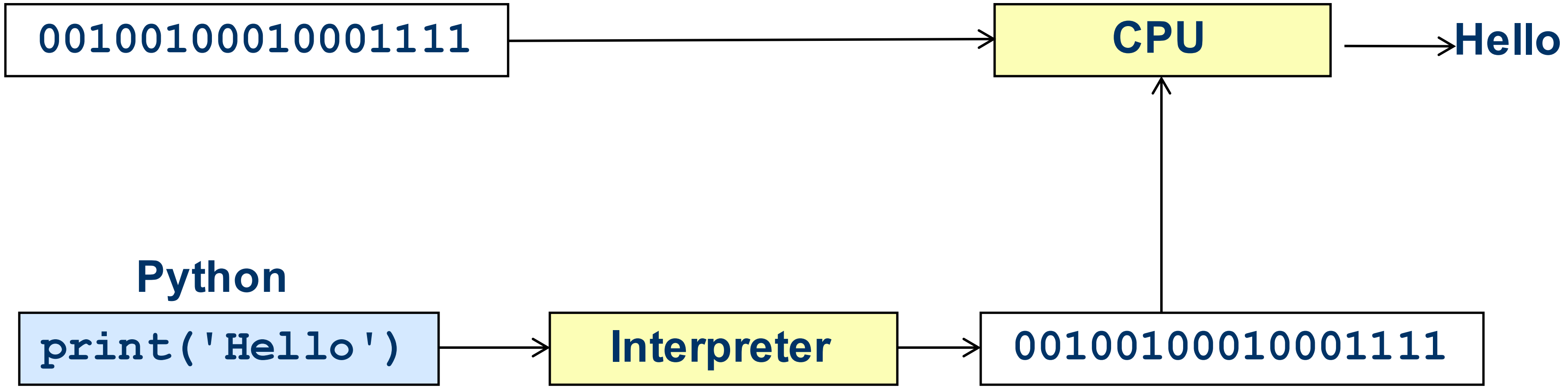
Solving Problems with Python

- Python is a portable high level language. Learning to write programs in Python provides you with:
  - Practice in logical thinking.
  - Practice in problem solving.

- What is a Programming Language?
  - The heart of a computer is the Central Processing Unit (CPU).
    - E.g. Pentium, Itanium, i3, i5, i7, etc.
  - The CPU:
    - is the part of the computer that actually runs programs.
    - responds to certain electrical signals in the form of binary numbers. E.g. 00101101
    - is designed to perform simple operations on pieces of data.

Solving Problems with Python

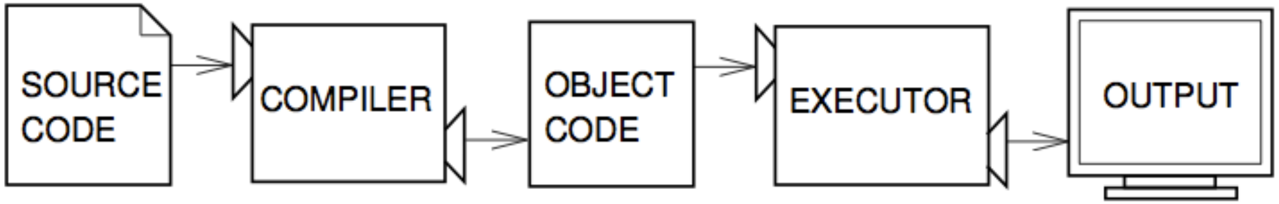
- A program is a set of instructions that a computer follows to perform a task.
- Instructions could be written in machine language (binary), but that is way too hard!



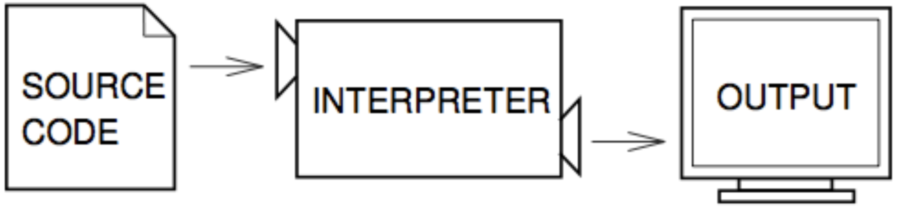
- A programming language makes more sense to people, but it must be translated to binary for the CPU.

Solving Problems with Python

- □ Translation is language-dependent. Use either a compiler or interpreter.
- □ **Compiler**: translates a high-level language program into a separate machine language program.
  - ● Machine language program can be executed at any time, without using the compiler.



- □ **Interpreter**: translates each high-level instruction to equivalent machine language instructions and immediately executes them.
  - ● Python uses an interpreter.
  - ● Interprets one instruction at a time.
  - ● No separate machine language program.



Hello World.py

```
# File:      hello_world.py
# Author:    your name
# Email Id:  your email id
# Version:   1.0  03 March 2014
# Description:  Displays the message Hello World.
#   This is my own work as defined by the University's
#   Academic Misconduct policy.
#

# Display Hello World message to the screen
print('Hello World')
```

Notice comment operator (#)

This produces the following output (Python Shell window):

```
>>>
Hello World
>>>
```

Python Programming Fundamentals

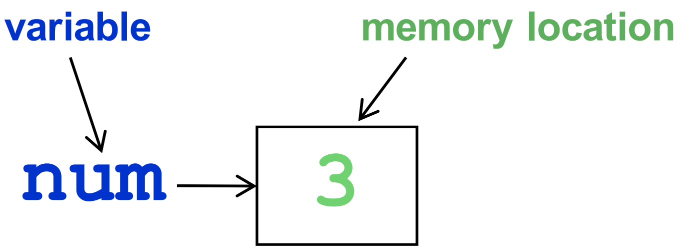
- **Comments #**
  - □ Span the entire line.
  - □ Are ignored by Python.
  - □ Describe each variable definition and to highlight the major parts of a program.
  - □ There should be a space between the # and the comment text. Comments should start with an uppercase letter and end with a period.

- **Variable**
  - □ A name that represents a value stored in the computer's memory.
  - □ Often want the values in these memory locations to be able to change.
  - □ We refer to these named memory locations as *variables*.
  - □ Python variables are automatically created when they are initialised.
  - □ Defined when they are assigned a value by using an assignment statement.
  - □ When a variable represents a value stored in memory, we say that the variable *references* the value.

Python Programming Fundamentals

- **Creating Variables**
  - □ Python allows you to assign a value to a variable.
  - □ Use an assignment statement to create a variable and make it reference a piece of data, for example:

```
>>> num = 3
```

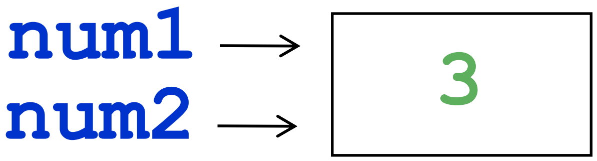


  - □ A variable named `num` will be created and it will reference the value 3.
  - □ We say that the `num` variable *references* the value 3.
  - □ This is not read as 'equals'.
  - □ Read as `num` is assigned a value of 3. Also referred to as *binding.*

Python Programming Fundamentals

- **Variable Reference vs Copy**
  - □ Python allows you to assign a value to a variable.
  - □ Whenever the variable is used, it takes the value by referring to it in memory, it does not take the value from a copy.
  - □ Consider the following example:

```
>>> num1 = 3
>>> num2 = num1
```



num1
num2
3

  - □ num1 and num2 now refer to the same object, i.e. the value 3.
  - □ In this case, Python uses referencing to memory.

Note! Understanding when references and copying is critical when working with large data sets, otherwise your program can produce unexpcted results. We will see examples as we go along.

Python Programming Fundamentals

- Variables and Constants

  - Variable
    - Measures a characteristic of interest that can take different values (e.g. height, width)
    - *adjective*. able or liable to change
    - *noun*. a name given to a memory location designed to store a data value (*Comp.Sci.*)
    - The value stored in it is likely to change during program execution

  - Constant
    - *noun*. a quantity that has a fixed value throughout a set of calculations
    - Called 'constant' as once set, the value stored does not/cannot change during program execution, e.g. pi = 3.14159.

Naming Variables, Python Keywords

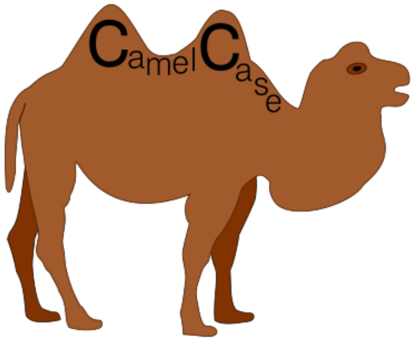
- All variable names must start with a letter (a–z, A-Z, or an underscore _).
- A variable name cannot contain spaces or punctuation @, $ and %.
- They may contain letters (a-z, A-Z), numbers (0-9) and underscores _ .
- Names are case sensitive:

count is different from Count

- Cannot use Python keywords as a variable name.

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

Variable Conventions



- lowercase for variable names.
- UPPERCASE for constants.
- Names should be short and descriptive.
- To represent names made up of multiple words, use either:
  - □ An underscore to separate words, e.g.: `game_status`
  - □ camelCase, e.g.: `gameStatus`

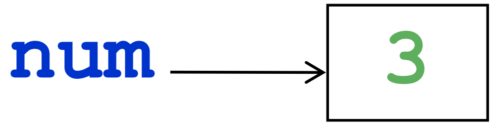
Be consistent in your approach!

- Class names start with an uppercase letter, e.g. `class Students`.
- A single leading underscore is for a private identifier, e.g. `_dictionary`
- Two leading underscores are used for a strongly private identifier, e.g. `__mydictionary`
- Also ending an identifier with two trailing underscores, the identifier is a language-defined special name.

Variable Reassignment

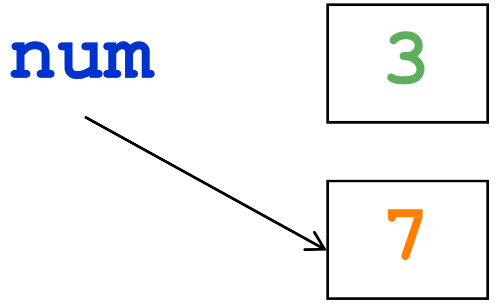
- Variables can reference different values during program execution.
- When you assign a value to a variable, the variable will reference that value until you assign it a different value.
- Example:

```
num = 3
```



- This creates the variable `num` and assigns it the value `3`

```
num = 7
```



- This assigns the value of `7` to `num`. The old value of `3` is still in the computer's memory, but can no longer be used.
- Garbage collection: when a value in memory is no longer referenced by a variable, the Python interpreter automatically removes it from memory.

Basic Data Types

- Numeric
  - □ **`int`** Integer – whole number with no decimal point. The size depends on the platform (32-bit vs 64-bit), e.g.

    `10, 100, -786,0b1001,0o11,0x69,-0x260` …

  - □ **`long`** large int values. Python automatically converts from `int` to `long`.

    `51924361L,  535633629843L, -4721885298529L, 0o122L` …

  - □ **`float`** double precision 64-bit real number – there is no `double float`.

    `0.0, 15.20, -21.9, 32.3+e18,` …

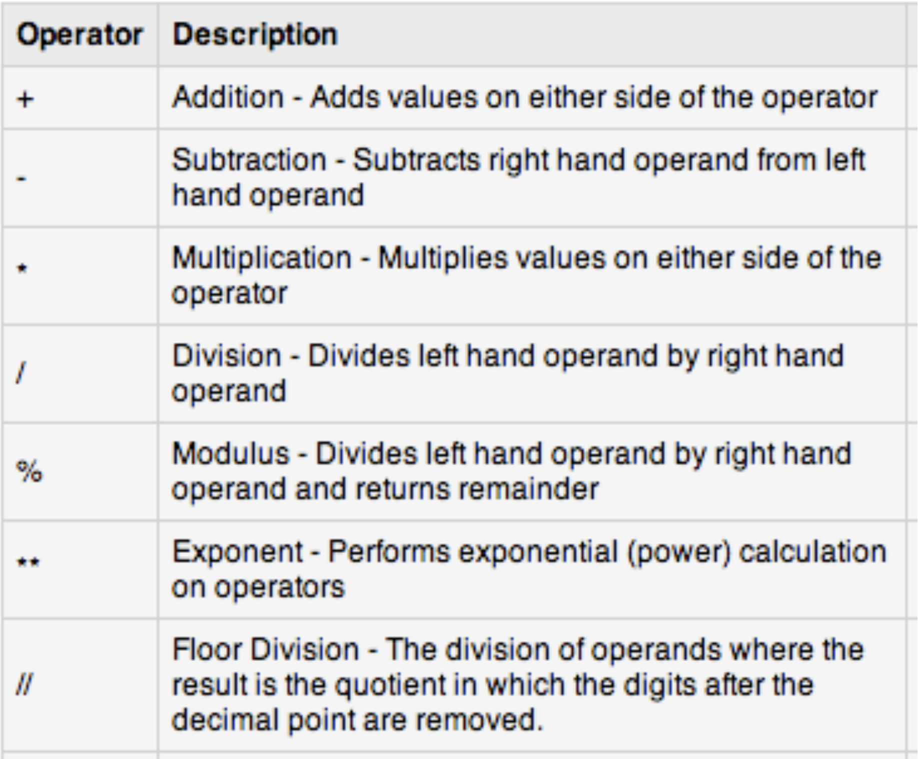
  - □ **`complex`** complex numbers

    `3.14j, 45.j, 9.322e-36j` …

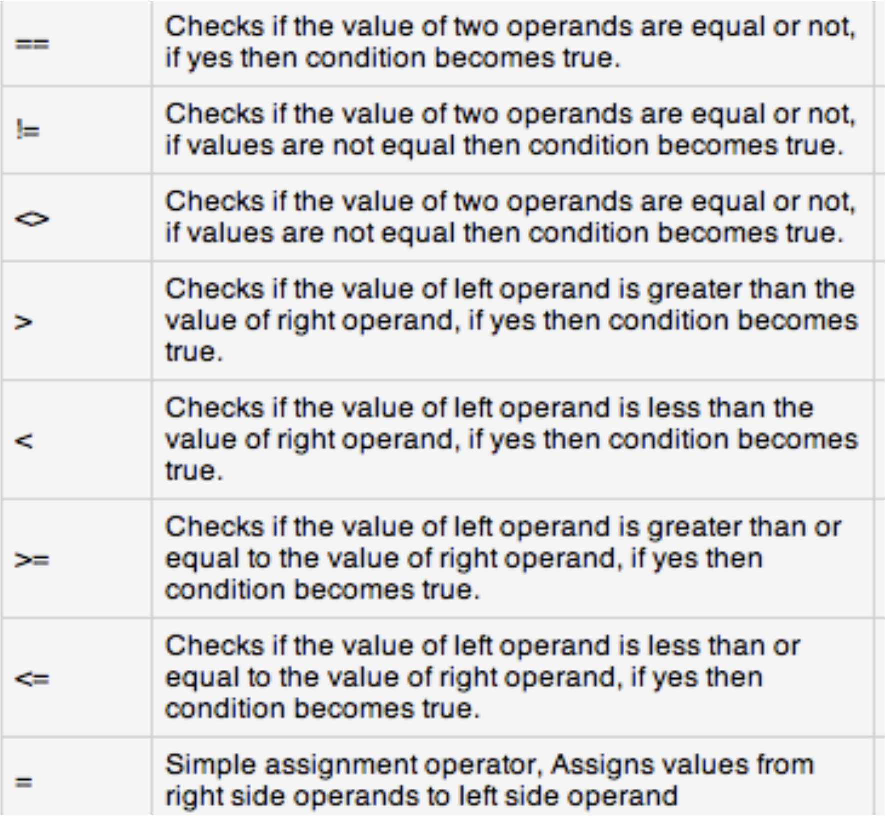
  - □ **`bool`** Boolean – reference one of two values: `True` or `False`.

Performing Calculations

- Expressions are calculated in a similar way as on your calculator
- An expression is a combination of values (or operands), variables, and operators. Note that variables may be used as operands.
  - Resulting value is typically assigned to a variable.



Operator	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
**	Exponent - Performs exponential (power) calculation on operators
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.



==	Checks if the value of two operands are equal or not, if yes then condition becomes true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.
<>	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.
=	Simple assignment operator, Assigns values from right side operands to left side operand

- A full list can be found on the course website (Python page)

Examples using Operators

- Assignment Statement =
  - □ A single equals sign (=) is called an assignment operator.
  - □ Causes the result of a calculation to be stored in memory
  - □ Examples:

```
num = 1+2  # “num is assigned the value of 1 plus 2”

# note the use of ;
num1 = 3; num2 = 5; total = num1+num2

# using a variable as an operand
num = 6
num = num + 1

# same as above but using the += operator
num = 6
num += 1
```

Precedence and Associativity

- Precedence and Associativity are used to determine how an expression is evaluated.

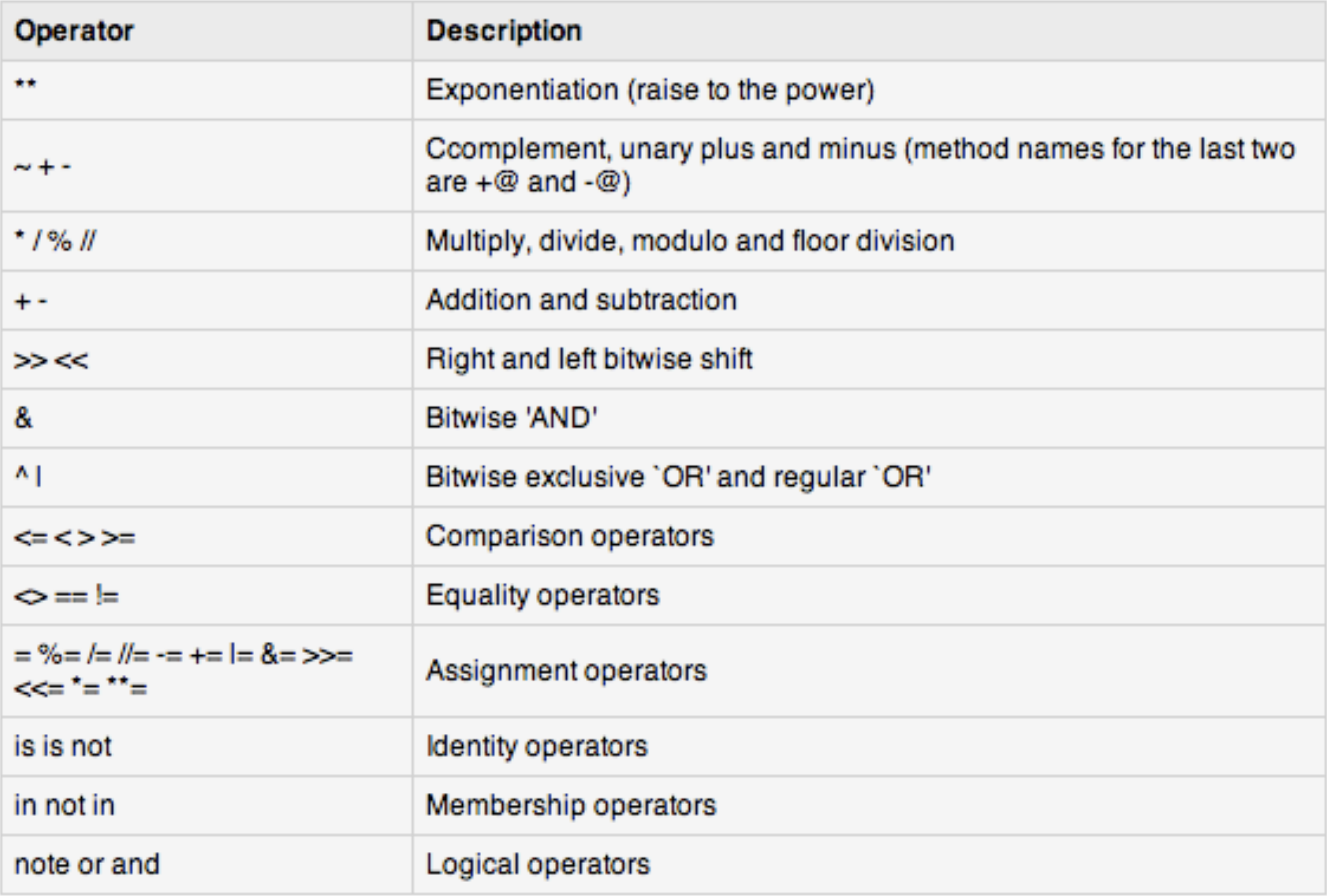
- Order of Operations: BODMAS
  - □ Python follows the standard algebraic rules for the order of operation (operator precedence):
    - ● **B**rackets (working from innermost to outermost)
    - ● **O**rders (exponentiation) (**)
    - ● **D**ivision (/, //), **M**ultiplication (*), Remainder (%) (left to right).
    - ● **A**ddition (+) and **S**ubtraction (-) operations (left to right).
  - □ Higher precedence is performed first.

Associativity

- Order of Operations
  - □ Operators on the same precedence level are evaluated left to right (associativity).
  - □ Example:

```
5*(2+2)  =
4*2/2*4  =
5*2+2  =
4*2/(2*4)  =
```

Order of precedence



Operator	Description
**	Exponentiation (raise to the power)
~ + -	Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += |= &= >>= <<= *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
note or and	Logical operators

A closer look at maths operators

- Division operators (/, //)
  - / operator performs floating point division.
  - // operator performs floor division.

```
 6.4 // 2 = 3.0
 6.8 // 2 = 3.0
-6.8 // 2 = -4.0
-6.4 // 2 = -4.0
```

Positive results are truncated, negative are rounded away from zero.

□ Exponent operator (**)
  - Raises a number to a power.

$x ** y = x^y$

□ Modulo operator (%)
  - Performs division and returns the remainder (modulus operator).

```
4 % 2 = 0
5 % 2 = 1
```

Programming Exercise

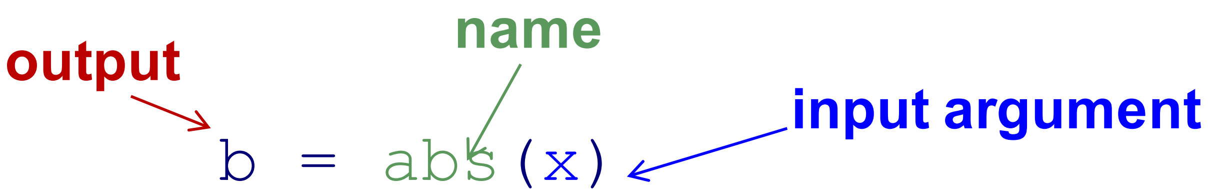
- Write a short program that assigns a value of 5 to a variable called height, a value of 10 to a variable width, and stores the result of width * height in a variable called area.
- The area should be displayed to the screen along with an informative statement.
- Solution

```
width = 10
height = 5
area = width * height
print(“The area of 5*10 is “,area)
```

The Python Standard Library

- Built-in functions – extensive library
  - □ The Python interpreter has a number of functions.
  - □ They are available without having to import a library.
  - □ Other functions can be obtained by using the command `import`.

- Functions consist of three components: name, input argument(s) and output, i.e.:



output

name

input argument

```
b = abs(x)
```

- The input (also called argument or parameter) goes inside the parentheses.

- The output is the returned value.

Built-in Python Functions

- Some functions require multiple inputs, e.g.

The `pow` function returns x to the power of y.

```
pow(x,y)
pow(2,3)
```

`pow(2,3)` calculates 2 to the power of 3 = 8

- You can assign a name to the output (variable):

```
result = pow(2,3)
```

Built-in Python Functions

- A few useful built-in functions:

Function	Description	Example

abs(x)	Returns the absolute value of x.	abs(-3)
		ans = 3

pow(x,y)	Returns x to the power y.	pow(5, 2)
		ans = 25

round(x[, n])	Returns the floating point value x rounded to n digits after the decimal point. If n is omitted, it defaults to zero.	round(1.6)
		ans = 2

int([number|string[, base]])	Converts a number or string to an integer. If no arguments are given, returns 0.	int('7')
		ans = 7

float([x])	Converts a string or number to floating point.	float('1.5')
		ans = 1.5

Built-in Python Functions

- A few useful built-in functions:

```
range([start], stop[, step])
```

Constructs progression of integer values, most often used in for loops. The arguments must be integers. The step value defaults to 1 if omitted. If the start argument is omitted, it defaults to 0.

```
str(x)
```

Converts object x to a string representation.

Built-in Python Functions

`input([prompt])`

If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that.

Examples:

```
guess = input('Please enter your guess: ')
name  = input('Please enter your name: ')
number = int(input('Enter number: '))
```

Built-in Python Functions

```
print([object, ...], sep=' ', end='\n', file=sys.stdout)
```

print(*object*(s) to the stream *file*, separated by *sep* and followed by *end*. *sep*, *end* and *file*, if present, must be given as keyword arguments.

Examples:

```
ans = 7
print("This is an example!")
print("This is another example:", 777)
print(ans)
print("The answer is:", ans)
print("i", "like", "donuts!")
print("i", "like", "donuts!", sep='-')
print("There are", ans, "donuts left...")
```

The `import` function

- In Python you can use any source file containing code by using the `import` statement.
- When the interpreter encounters an import statement, it imports the module if the module is present in the search path.
- `import` has the syntax

```
import module1[,module2,…,moduleN]
```

  where a module is simply a .py file.

- One module used often is `math` (mathematical functions):
  - □ Provides access to the mathematical functions.
  - □ Is imported at the top of your program using the statement:

```
import math
```

math Modules

Function	Description
math.ceil(x)	Returns the ceiling of x, the smallest integer greater than or equal to x.
math.floor(x)	Returns the floor of x, the largest integer less than or equal to x.
math.sqrt(x)	Returns the square root of x.
math.cos(x)	Returns the cosine of x radians.
math.sin(x)	Return the sine of x radians.
math.tan(x)	Returns the tangent of x radians.
math.degrees(x)	Converts angle x from radians to degrees.
math.radians(x)	Converts angle x from degrees to radians.
math.pi	The mathematical constant pi.

The `random` Module

- It provides access to random number generator.
- Is imported at the top of your program using the statement:

```
import random
```

Function	Description
random.randint(a, b)	Returns a random integer N such that a <= N <= b.
random.choice(seq)	Returns a random element from the non-empty sequence seq.
random.random()	Returns the next random floating point number in the range 0.0, 1.0.
random.shuffle(x)	Shuffle the sequence x in place.

Comparison and Boolean Operators

- Comparison/relational operators are used to compare the values of two objects.

- Boolean operators allow us to combine comparisons.
    - □ Used to make decisions
    - □ Used in expressions where a True/False result is required.

- Both are useful in control structures.

==	Is equal to
!=	Is not equal to
<	Is less than
<=	Is less than or equal to
>	Is greater than
>=	Is greater than or equal to

Example: Comparison Operators

- Consider the following:

```
>>> x = 5
>>> y = 1
>>> x < y
False
```

- The result of comparison is either `True` or `False`.

- The answer may be used in selection statements and repetition statements to make decisions.

Boolean Operators and, or, not

- The Boolean Operators `and`, `or`, `not` combine comparisons.
  - □ Used when either one, or both, of two (or more) conditions must be satisfied.

```
IF (it is 1pm and I am hungry) THEN
        have lunch

x >= 0 and x <= 5            ⇨  and
x <= 2 or x >= 4             ⇨  or
not a == b and c == d        ⇨  not
```

- `x` **`and`** `y:` first evaluates *x*. if *x* is false, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

- `x` **`or`** `y:` first evaluates *x*; if *x* is true, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

Aside: Boolean Operators and Truth Tables

x	not x
T	F
F	T

not

x	y	and
F	F	F
F	T	F
T	F	F
T	T	T

and

x	y	or
F	F	F
F	T	T
T	F	T
T	T	T

or

Exercise: Comparison Operators

```
>>> x = 5
>>> y = 1
>>> z = 5
>>> res = x < y
>>> res
```

```
>>> res = x == z
>>> res
```

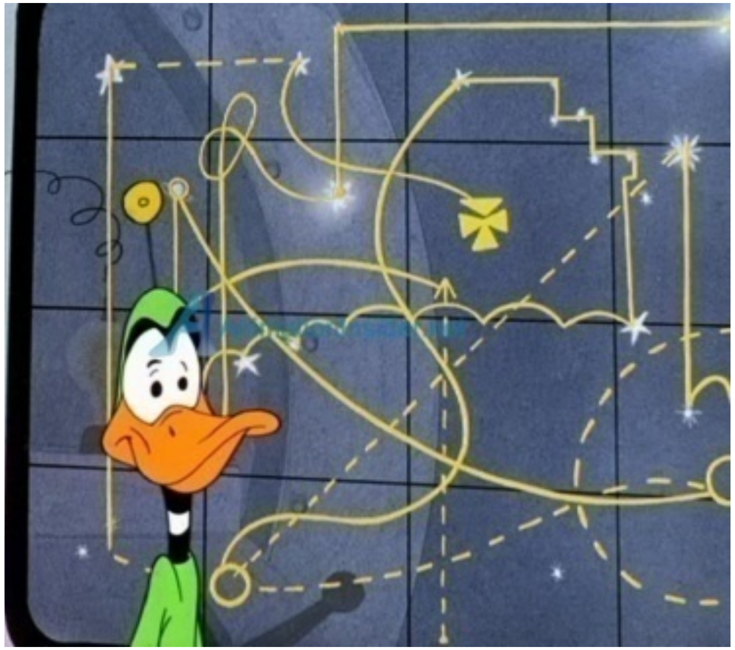
```
>>> res = y < x and z < y
>>> res
```

```
>>> res = y < x or y == z
>>> res
```

Up next …

Control Flow and Strings

- Selection:
  - □ if, elif and else
  - □ if-else, if-elif-else
  - □ Selectively running sections of code (sequences of commands).



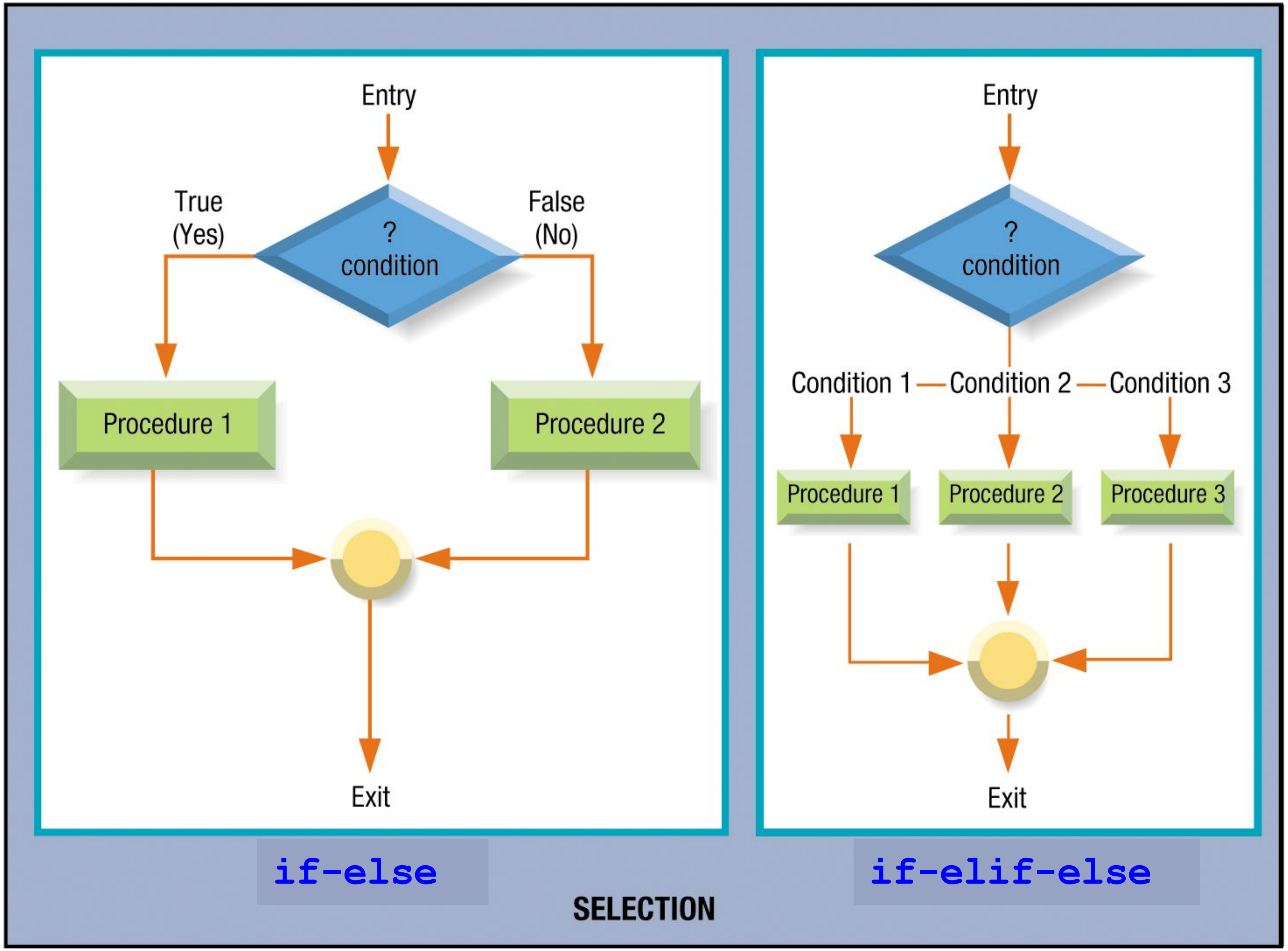
- Repetition:
  - □ for iterating over a collection of items.
  - □ while repeatedly executing commands for some TRUE condition.
- Altering control flow:
  - □ Break, Continue and Pass
- String operations

Control Structures

- Control structures come in the flavours:
  - □ **Selection** (making decisions)
    - ● if
    - ● if-else
    - ● if-elif-else

  - □ **Repetition** (doing the same thing over and over)
    - ● while
    - ● for
  - □ **Sequence** statements are executed one after the other in order. Can occur in blocks inside of selection or repetition structures.

Selection Structures



The `if` Selection Structure

- A simple if statement has the following form:

```
if expression:          ←—— Boolean expression
    <do something>
```

- If the expression is `True`, the statements indented under the if statement are executed.

- If the expression is `False`, the program jumps immediately to the statement following the indented block.

- **Important!** must use consistent indentation – the start and end of blocks are indicated by indentation. Python uses indentation to signify its block structure.

```
if age >= 18:
    print('You can vote!')
```

Exercise: `if` statement

- Write code to:
  - □ Generate a random number between 1 – 10 and then ask (prompt) the user to guess the random number. Display the user's guess to the screen and the random number to the screen, using the phrases shown below.

```
You guessed:  <user's guess>
The random number is: <random number>
```

  - □ If the user guesses correctly, display:

```
Well done – you guessed it!
```

The `if-else` Selection Structure

- Allows us to execute one set of statements if the expression is `true` and a different set if the expression is `false`:

```
if expression:
    <do something>              ←  sequence
else:
     <do something else>        ←  sequence
```

```
if mark >= 50:
   print('You passed!')
else:
   print('You failed!')
```

Note the indentation – take care to use this style of indenting.

Exercise: `if-else` statement

- Recall the random number generator program. Write code to:
  - □ Generate a random number between 1 – 10 and then ask (prompt) the user to guess the random number. Display the user's guess to the screen and the random number to the screen, using the phrases shown below.

```
You guessed:  <user's guess>
The random number is: <random number>
```

  - □ If the user guesses correctly, display the congratulations message below, otherwise display the commiserations message.

```
Well done - you guessed it!

Too bad - better luck next time!
```

Combining Selection Structures

- **and** – true only if all tested conditions are true. E.g. if temperature is greater than zero and less than 40

```
if temperature > 0 and temperature < 40:
  print('Normal temperature')
else:
  print('Extreme temperature!')
```

- **or** – true if either (any) of the conditions are true. E.g. if it is raining or highUV

```
raining = True
highUV = True
if raining or highUV:
  print('Take an umbrella!')
```

The `if-elif-else` Selection Structure

- Nested `if-else` statements may be difficult to read. The `if-elif-else` statement allows you to check multiple criteria while keeping the code easy to read, e.g.:

```
if mark >= 85:
   print('HD')
elif mark >= 75:
   print('D')
elif mark >= 65:
   print('C')
elif mark >= 55:
   print('P1')
elif mark >= 50:
   print('P2')
elif mark >= 40:
   print('F1')
else:
   print('F2')
```

Exercise: `if-elif-else` statement

- Generate a random number between 1 – 10 and then ask (prompt) the user to guess the random number. Display the user's guess to the screen and the random number to the screen, using the phrases shown below.

```
You guessed:  <user's guess>
The random number is: <random number>
```

- If the user guesses correctly, display:

```
Well done - you guessed it!
```

If the user guesses lower than the random number, display: `Too low!`

If the user guess higher than the random number, display: `Too high!`

Ternary Expressions

- A ternary expression allows you to combine an `if-else` block into a single line or expression.

- The synatx is:

```
value = true-expr if condition else false-expr
```

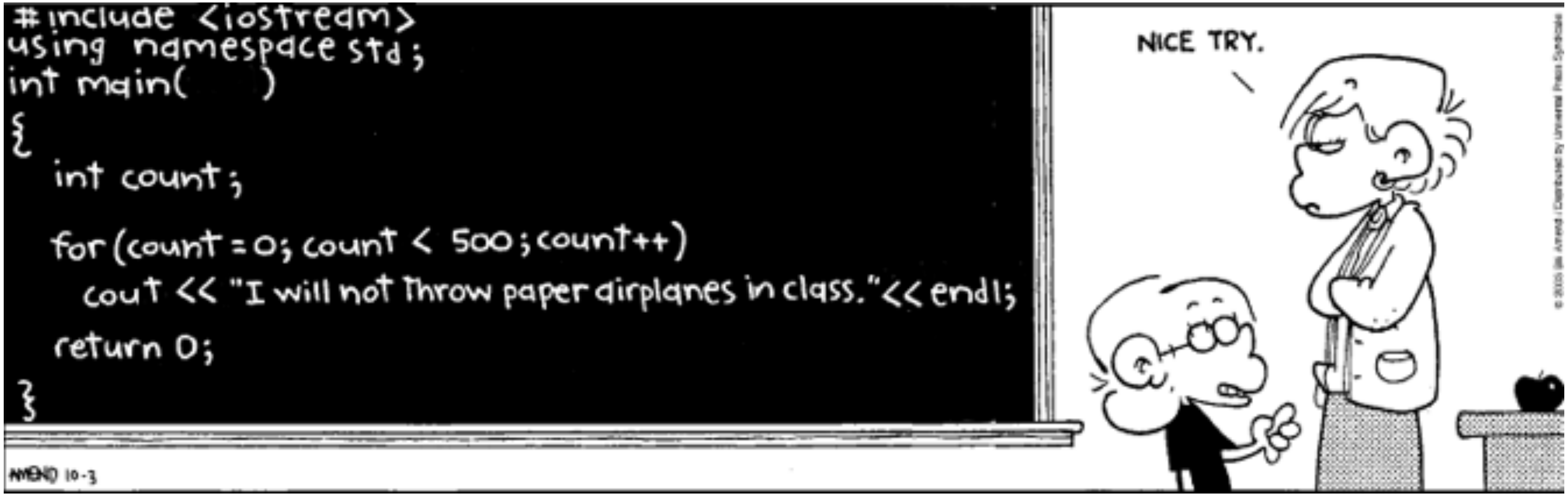
- Example:

```
mark = 88
'High Distinction' if mark >= 85 else 'Other Grade'
```

- Note only one expression will be evaluated.

- Use wisely! Compactness vs readability of code is a consideration.

Repetition Structures (`while`)



```
#include <iostream>
using namespace std;
int main(    )
{
  int count;

  for(count=0; count < 500; count++)
    cout << "I will not Throw paper airplanes in class." << endl;

  return 0;
}
```

NICE TRY.
AMEND 10-3
© 2005 Bill Amend / Distributed by Universal Press Syndicate

Repetition structures

- Loops: used when you need to repeat a set of instructions multiple times. Two types of loops:

- `while` loop
  - ☑ Good choice when you do not know, in advance, how many times the set of instructions should be repeated.
  - □ Instructions are repeated based on a condition being evaluated as either TRUE or FALSE.

- `for` loop
  - ☑ Used when we have a set of instructions we want to repeat *n* times, where *n* can usually be worked out in advance.
  - □ *n* is finite and often countable in advance.
  - □ The value of *n* can be either user-supplied or calculated from a data structure in the program itself.
  - □ For loops can be written as while loops, but the reverse is not true!

Which Loop To Use?

- Example: I want to print the numbers 1 to 10 to the screen.
  - Use a for loop (n = 10)

- Example: I want to print the statement “The Weather is Fine!” for as long as the temperature is between 20 and 30 degrees.
  - Use a while loop (need to evaluate a TRUE/FALSE situation to decide when to exit the loop).

- Example: I want to print the days of the week.
  - Use a for loop - we can figure this out in advance!

- Example: I want to read data from a file.
  - Use a while loop - I can’t easily work out the number of lines in a file before I read it in!

Control Structures: `while` loop

- A `while` loop has the following form:

```
while expression:
    <do something while true>
else:
    <do something else>
```

- The `else` clause is optional.

- As long as the `expression` is `True`, the while block will execute the associated set of commands.

- If the `expression` is or becomes `False`, the loop terminates. If the optional `else` clause is present, the alternative is executed.

Example: while loop

```
a = 0                      # initialise a
while a < 10:              # loop while a less than 10
    print(a)               # output value of a
    a = a + 1              # increment a
else:                      # otherwise
    print('done')          # print finishing sentence
```

- **Important!** Remember to use consistent indentation.
- The start and end of blocks are indicated by indentation.
- Python uses indentation to signify its block structure.
- Question: how else could you have written `a = a + 1`?

Control Structures: `while` loop

- What is the output produced by the following code?

```
a = 0
while a < 3:
    print('Going loopy', a)
    a += 1

print('Done!')
```

sequence

- Question: what happens if you don't indent?

Control Structures: `for` loop

- A `for` loop has the following form:

```
for iterating variable in sequence:
    <do something>
```

- The `for` loop behaves much like a *foreach* operator – i.e. it applies the subsequent code sequence to each member of a collection, e.g.:

```
fruits = ['banana', 'apple',  'mango']
for i in range(len(fruits)):
   print 'The current fruit is :', fruits[i]

print('The fruit bowl is empty!')
```

- What do `range` and `len` do?
- For very long ranges, `xrange` is recommended instead.

Examples: `for` loop

- Some more examples of a `for` loop:

```
# count forwards
for i in range(10):
    print(i)

# count backwards
for i in range(10,0,-1):
    print(i)

# counting by twos (forwards)
for i in range(0,20,2):
    print(i)

# counting by twos (backwards)
for i in range(20,0,-2):
    print(i)
```

The String Data Type

- As we have seen, we can create a string by enclosing characters in quotes.

- Python treats single quotes the same as double quotes.

- Strings are immutable. Once they are set, they cannot be changed.

- Strings are made up of individual characters, which may be accessed via slicing.

- A string consists of a sequence of characters.
  - □ The first character of a string “hey” is 'h', the second character s 'e', etc.

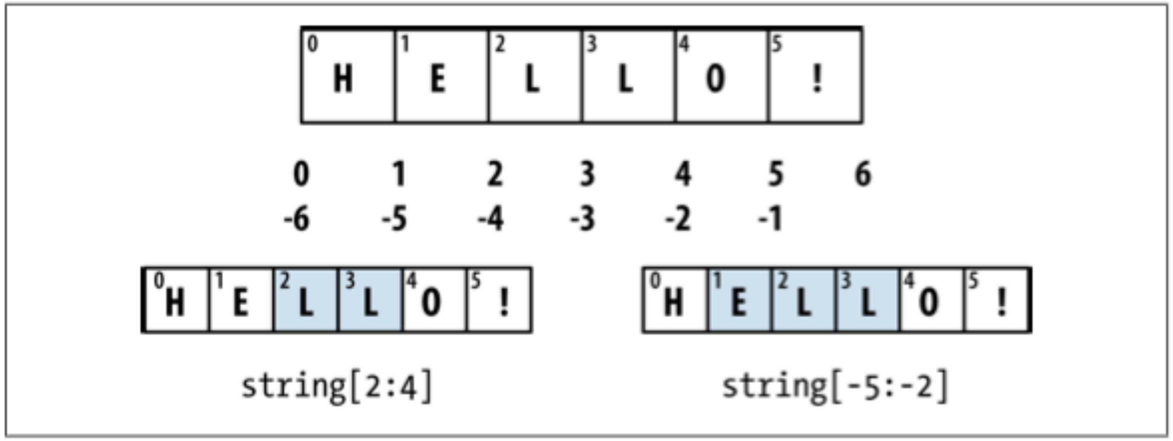
Strings and slicing

- String: can be enclosed in single ' ' or double “ “ quotes, e.g.

```
>>> str = 'Hello!'                 # or you could use ‶ ‶
>>> print(str)                     # Prints complete string
```

- Can slice a string by using index numbers:

```
>>> print(str[0])                  # Prints first character
>>> print(str[2:4])                # Prints 3rd to 4th characters
>>> print(str[2:])                 # Prints 3rd character on
>>> print(str[-1])                 # Prints from right to left
>>> print(str * 2)                 # Prints string twice
>>> print(str + ‶ Program")        # Prints concatenated string.
```



More on strings…

- Creating and assigning strings:

```
>>> str1 = 'string one'
>>> str2 = "this is a string"
>>> str3 = 'this is another string'
>>> print(str2)
this is a string
>>> str3
'this is another string'
```

s	t	r	i	n	g		o	n	e
str1[0]	str1[1]	str1[2]	str1[3]	str1[4]	str1[5]	str1[6]	str1[7]	str1[8]	str1[9]

More on strings…

s	t	r	i	n	g		o	n	e
str1[0]	str1[1]	str1[2]	str1[3]	str1[4]	str1[5]	str1[6]	str1[7]	str1[8]	str1[9]

- Accessing values (characters and substrings) in Strings:
  - □ Use square brackets for slicing along with the index or indices to obtain a substring.
  - □ We can access values using:

str1[start]	Value at index start. Eg: str1[2] has value 'r'
str1[start:end]	Slice from start to end-1. Eg: str1[2:6] is 'ring'
str1[start:end:step]	From start to end-1 with step size. Eg: str1[0:5:2] is 'srn'

More on strings

- Comparison operators and strings:
  - □ Strings are compared lexicographically (ASCII value order).

```
>>> str1 = "abc"
>>> str2 = "abc"
>>> str3 = "xyz"

>>> str1 < str2
False

>>> str1 == str2
True

>>> str1 < str3
True

>>> str1 > str3
False
```

```
>>> str1 == str3
False

>>> str1 != str3
True

>>> str1 < str3 and str2 ==
"abc"
True
```

More on strings

- The `in` operator:
  - □ Useful for checking membership.
  - □ Returns a Boolean value – `True` or `False`. For example:

```
>>> str1 = "aeiou"

>>> 'a' in str1
True

>>> 'z' in str1
False

>>> if 'u' in str1:
        print("It's a vowel!")

It's a vowel!
```

More on strings

- Concatenation (+) operator
  - □ Create new strings from existing ones.

- Repetition (*) operator
  - □ Creates new strings, concatenating multiple copies of the same string.

```
>>> str1 = 'hi '
>>> str2 = 'there...'

>>> str1 + str2
'hi there...'

>>> str1 * 3
'hi hi hi '
```

More on strings…

- Updating strings:
  - □ Strings are immutable: changing an element of a string requires creating a new string.
  - □ Cannot update individual characters or substrings in a string.
  - □ Update an existing string by (re)assigning a variable to another string.

```
>>> str1 = "this is"
>>> str2 = "a string"
>>> str1 = str1 + "something different"
>>> print(str1)

this issomething different


>>> str1 = "different altogether"
>>> print(str1)

different altogether
```

More on strings…

- String built-in functions:
  - `len()`: returns the length of a string.
  - `max()`: returns the max alphabetic character.
  - `min()`: returns the min alphabetic character.

```
>>> str1 = "roger"
>>> len(str1)
5

>>> max(str1)
'r'

>>> min(str1)
'e'
```

More on strings…

- String methods:
  - □ Strings are objects, i.e. contain data and processing for data.
  - □ Processing via methods that are called using dot notation.

  - □ Example: if `str1` is a string, then

```
str1.upper()
```

is a method that returns a string that has the characters of `str1` converted to upper case.

```
>>> str1 = 'this is fun'
>>> str1.upper()

'THIS IS FUN'
```

More on strings…

- Some useful string methods:
  - □ `string.lower()`

    converts all uppercase letters in string to lowercase.
  - □ `string.upper()`

    converts all lowercase letters in string to uppercase.
  - □ `string.find(t, start, end)`

    find string t in string or in substring if given [start:end].
  - □ `string.split(sep, maxsplit)`

    returns a list of the words in the string, using sep as the delimiter string (default is space). If maxsplit is given, at most maxsplit splits are done.

  - □ See Python docs (http://www.python.org/doc/) for all string methods.

More on strings…

- Some programming tasks require you to access the individual characters in a string.
- Iterating over a String with the while loop:
  - □ Access the individual characters in a string with an index. Each character has an index that specifies its position in the string.
  - □ We need to use the `len()` function:

```
str1 = 'the blacklist'
index = 0
while index < len(str1):
    print(str1[index])
index = index + 1
```

What happens if I add a "," at the end of the print statement?

  - □ `len()` prevents the loop from iterating beyond the end of the string. The loop iterates as long as `index` is less than the length of the string.

Up next …

More Data Structures

- Tuples
  - □ The fixed-length immutable object
- Lists
  - □ The variable-length mutable object
  - □ Concatenating, Sorting and Slicing
- Dict
  - □ The variable-size mutable hash map
- Sets: mathematical sets
- List, Set and Dict Comprehensions

"Your recent Amazon purchases, Tweet score and location history makes you 23.5% welcome here."

The Tuple() data type

- `Tuple()`
  - One-dimensional, Immutable, Fixed-length.
  - Predominantly used as read-only.
  - Lightweight – useful as they are computationally cheap since they provide only minimal operations = FAST
  - Useful when information remains the same (e.g. months of the year).
  - Correspond to mathematical tuples.

- Contains a sequence of Python objects

- Can create just by writing a set of objects separated by a comma:

```
>>> easy_tuple = 4,5,6
>>> easy_tuple
(4,5,6)
```

Nested Tuples

- Can define tuples within tuples.
- In this case, will need to use parentheses () to indicate nesting levels, e.g.

```
>>> nested_tup = (4,5,6),(7,8)
>>> nested_tup

((4,5,6),(7,8))
```

- Can use `tuple` as a command, to convert any object into a tuple, e.g.

```
>>> tup = tuple('string')
>>> tup

('s', 't', 'r', 'i', 'n', 'g')
```

- Elements can be accessed with square brackets [], e.g.:

```
>>> tup[0]
```

Tuples and Immutability

- Once a tuple is defined, it cannot be changed, for example

```
>>> tup = tuple(['foo', [1, 2], True])
>>> tup[0]

    'foo'                # returns what we expect


>>> tup[2] = False       # cannot change the value!



Traceback (most recent call last):
  File "<pyshell#74>", line 1, in <module>
    tup[2] = False
TypeError: 'tuple' object does not support item assignment
```

Immutability Workarounds

- Can concatenate tuples using the + operator:

```
 >>> (4, None, 'foo') + (6, 0) + ('bar',)
(4, None, 'foo', 6, 0, 'bar')
```

- Can also use the replication operator * on tuples.

- May not have the expected effect!

```
>>> ('foo', 'bar')*4
 ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

More on Tuples

- If a tuple contains a mutable object, we can modify this object.
- Warning: cannot move the mutable object – its location is fixed!

```
>>> tup = tuple(['foo', [1, 2], True])
>>> tup[1].append(3)                    # append 3
>>> tup                                 # what's in tup now?
('foo', [1, 2, 3], True)
```

- Tuple methods: not many, since tuples are not modifiable. The method `count` is useful:

```
>>> a = (1, 2, 2, 2, 3, 4, 2)
>>> a.count(2)                          # what does this do?
4
```

- Typing `dir(())` gives a full list of tuple methods (`count`, `index`).

Unpacking Tuples

- If we try to assign a tuple to a number of variables, Python will 'unpack' the tuple, e.g.

```
>>> tup = (4, 5, 6)
>>> a, b, c = tup
>>> b
5                                  # a contains 4 and c contains 6

>>> tup = 4, 5, (6, 7)
>>> a, b, (c, d) = tup             # we specify the same structure
>>> d
7

>>> a, b, c = tup                  # we keep the nested structure
>>> c
(6, 7)
```

The List [] Data Type

- List[]
  - □ A sequence of zero or more objects.
  - □ Supports the same indexing and slicing as a string.
  - □ Define using square brackets []
  - □ Variable length and Mutable: can be modified, e.g. Lists can be populated, empty, sorted, and reversed. Individual or multiple items can be inserted, updated, or removed.

```
>>> a_list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
>>> tinylist = [123, 'john']

>>> print(a_list[0])          # Prints first element of the list
>>> print(a_list[1:3])        # Prints second elements 2 and 3
>>> print(a_list[2:])         # Prints elements from 3rd element on
>>> print(a_list[i:j:k])      # Prints from i to j in steps of k
>>> print(a_list[::n])        # Prints every nth element
>>> print(tinylist * 2)       # Prints list twice
>>> print(list + tinylist)    # Prints concatenated lists
```

Indexing Lists

- Consider the following

```
>>> List_1 = [1, 'abc', 2, 'def', 3, 'ghi']
>>> List_1
[1, 'abc', 2, 'def', 3, 'ghi']

>>> List_1[2:3] = [4,'jkl']                # Lists are mutable!
>>> List_1
[1, 'abc', 4, 'jkl', 3, 'ghi']
```

List_1

1	'abc'	2	'def'	3	'ghi'	
0	1	2	3	4	5	forward index
-6	-5	-4	-3	-2	-1	backward index

Examples: Slicing List Items

```
>>> List_2 = [11, 12, 13, 14]
>>> List_2 [1:3]
[12,13]
```

```
>>> List_2 [:3]
[11, 12, 13]
>>> List_2 [-1]
14
```

```
>>> List_3 = [1, 'abc', 2, 'def', 3, 'ghi']
>>> List_3[2:]
[2, 'def', 3, 'ghi']
```

```
>>> List_3[::2]
[1, 2, 3]
```

# Print every second item. Selects items in positions that are a multiple of 2. 2*0=0 is considered a multiple of 2 …

```
>>> List_3[1::2]
['abc', 'def', 'ghi']
```

# Print every second item starting from **location [1]**

```
>>> List_3[::-1]
['ghi', 3, 'def', 2, 'abc', 1]
```

# Quick list reversal!

List Operations: Membership

- We can use membership to check whether an object is a member of a list.

- We use in and not in, e.g.

```
>>> List_4 = ['one', 'two', 'three']
>>> 'one' in List_4
True

>>> 'four' in List_4
False

>>> 'four' not in List_4
True

>>> if 'one' in List_4:
    print('Found it!')
Found it!
```

Methods to Modify the List: `append()`

- You can add elements to a list with the `append()` method.

```
>>> List_5 = [1, 2, 3, 5, 6, 7, 8, 9, 10]
>>> List_5.append(11)
>>> List_5
[1, 2, 3, 5, 6, 7, 8, 9, 10, 11]

>>> List_5.append('twelve')
>>> List_5
[1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 'twelve']

>>> List_5.append([13,14])
>>> List_5
[1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 'twelve',[13,14]]
```

- Note that `append()` will add the second list as a single list element to the end of a list *(try typing `List_5[10]` and see what comes back!).*

Methods to Modify the List: `extend()`

- The `extend()` method will extend the current list by adding each item as a separate element, even if the new item is a list.

```
>>> List_5 = [1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 'twelve']
>>> List_5.extend([13,14])
>>> List_5
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 'twelve',13, 14
```

versus…

```
 # Using append()
[1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 'twelve',[13,14]]
```

- The use of `extend()` over + is preferred, as `extend` is faster.

Methods to Modify the List: `insert()`

- We can insert value(s) at the element in the list with index `m`, using the syntax `insert(m, value)`, e.g.:

```
>>> List_7 = [5, 6, 7, 8]
>>> List_7
[5, 6, 7, 8]

>>> List_8 = ['one', 'two', 'three', 'four']
>>> List_8
['one', 'two', 'three', 'four']

>>> List_8.insert(2,List_7)
>>> List_8
['one', 'two', [5, 6, 7, 8], 'three', 'four']
```

Methods to Modify the List: `del()`

- We use `del()` to remove elements from a list by specifying the position of these elements:

```
>>> List_5 = [1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 'twelve']
[1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 'twelve']

>>> del List_5[3]
>>> List_5
[1, 2, 3, 6, 7, 8, 9, 10, 11, 'twelve']

>>> del List_5[1:4]
>>> List_5
[1, 7, 8, 9, 10, 11, 'twelve']

>>> List_5[2:4]  = []
>>> List_5
[1, 7, 10, 11, 'twelve']
```

The `del()` statement is preferred method of deleting values from a list – it is easier to read than the equivalent setting of a slice to an empty list.

Methods to Modify the List: remove()

- If we want to remove a value (not a value in a specified position), then we use remove(), e.g.:

```
>>> List_6 = [1, 3, 6, 3, 8, 9, 'ten']
>>> List_6.remove(8)
>>> List_6
[1, 3, 6, 3, 9, 'ten']                      # 8 has been removed

>>> List_6.remove('ten')
>>> List_6
[1, 3, 6, 3, 9]                             # 'ten' has been removed

>>> List_6.remove(3)
>>> List_6
                                            # 3 has been removed
[1, 6, 3, 9]                                (only the first instance)

>>> List_6.remove(0)
… will cause an error!
```

Other List Methods

- Typing `dir([])` at the chevron `>>>` will provide a full list.
- Ignore those methods containing “__” in the name. We are interested in:

```
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
 'reverse', 'sort']
```

- E.g. reverse the order of the values in the list.

```
>>> List_9 = [1, 2, 3, 4]
>>> List_9
[1, 2, 3, 4]
>>> List_9.reverse()
>>> List_9
[4, 3, 2, 1]
```

- E.g. sort the values in a list.

```
>>> List_10 = [4, 6, 1, 9]
>>> List_10
[4, 6, 1, 9]

>>> List_10.sort()
>>> List_10
[1, 4, 6, 9]

>> List_10.sort(reverse=True)
```

Non-modifying List Methods

- These methods *do not* change the list.
- E.g. `index(), len(), max(), min()`

```
>>> List_6 = [1, 3, 6, 3, 8, 9, 10]
>>> List_6.index(8)
4

>>> List_6.index(3)
1

>>> len(List_6)
7

>>> max(List_6)
10

>>> min(List_6)
1
```

Other cool methods

- The method `enumerate()` is quite useful:

```
# Original code
collection = [1,2,3,4]
i= 0
for value in collection:
       print(i)
       i += 1

# Using enumerate()
collection = [1,2,3,4]
for i, value in enumerate(collection):
       print(i)
```

Advanced zipping using *

- We can use zip to unzip …

```
>>> actors = [('Leo', 'DiCaprio'), ('George', 'Clooney'),
        ('Daniel','Craig')]

>>> first_names, last_names = zip(*actors)
>>> first_names
('Leo', 'George', 'Daniel')

>>> last_names
('DiCaprio', 'Clooney', 'Craig')
```

Other cool methods

- The method `zip()` is also quite nice:

```
>>> seq1 = ['foo', 'bar', 'baz']
>>> seq2 = ['one', 'two', 'three']

>>> zip(seq1, seq2)
[('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

- `zip()` is commonly used with `enumerate()`:

```
for i,(a, b) in enumerate(zip(seq1, seq2)):
        print('%d: %s, %s' % (i, a, b))
```

yields:

```
0: foo, one
1: bar, two
2: baz, three
```

Printing requires the use of special formatting. We will look at this separately – for now, just go with it!

The Dict{} Data Type

- Dict{}
  - □ A hash table type. Used as an associative array.
  - □ Variable size and mutable.
  - □ Format is key:value

```
>>> dict_1 = {'name':'john','code':6734,'dept':'sales'}
>>> dict_1                           # Prints complete dictionary
>>> dict_1.keys()                    # Prints all the keys
>>> dict_1.values()                  # Prints all the values

>>> dict_2 = {'a':'belinda','b':[1,2,3,4]}
>>> dict_2['a']
'belinda'

>>> dict_2[3] = 'lucky number'
>>> dict_2
{'a': 'belinda', 3: 'lucky number', 'b': [1, 2, 3, 4]}
```

The `Dict{}` Data Type

- Typing `>>>dir({})` will provide a full list of methods:

```
['clear', 'copy', 'fromkeys', 'get', 'has_key', 'items',
 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop',
 'popitem', 'setdefault', 'update', 'values', 'viewitems',
 'viewkeys', 'viewvalues']
```

- E.g.:

```
>>> dict_2 = {'a':'belinda','b':[1,2,3,4]}
>>> dict_2[5] = 'some value'
>>> dict_2['dummy'] = 'another value'
>>> dict_2
{'a': 'belinda', 'dummy': 'another value', 'b': [1, 2, 3, 4], 5:
'some value'}

>>> del dict_2[5]
>>> val = dict_2.pop('dummy')
```

More `Dict{}` Examples

```
>>> mapping = dict(zip(range(5), reversed(range(5))))
>>> mapping
{0: 4, 1: 3, 2: 2, 3: 1, 4: 0}



>>> words = ['apple', 'bat', 'bar', 'atom', 'book']
>>> by_letter = {}
>>> for word in words:
        letter = word[0]
        if letter not in by_letter:
                by_letter[letter] = [word]
        else:
                by_letter[letter].append(word)

>>> by_letter
{'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

The `Set{}` Data Type

- `Set{}`
    - An unordered collection of unique elements
    - Thought of as the 'keys' of a dict type
    - Support mathematical set functions, e.g. union, intersection.
    - Create using `{}` or the `set()` function

```
>>> set([2, 2, 2, 1, 3, 3])
set([1, 2, 3])

>>> {2, 2, 2, 1, 3, 3}
set([1, 2, 3])
```

The Set{ } Data Type

- Some examples:

```
>>> a = {1, 2, 3, 4, 5}
>>> b = {3, 4, 5, 6, 7, 8}

>>> a | b # union (or)
set([1, 2, 3, 4, 5, 6, 7, 8])

>>> a & b # intersection (and)
set([3, 4, 5])

>>> a - b # difference
set([1, 2])

>>> a ^ b # symmetric difference (xor)
set([1, 2, 6, 7, 8])
```

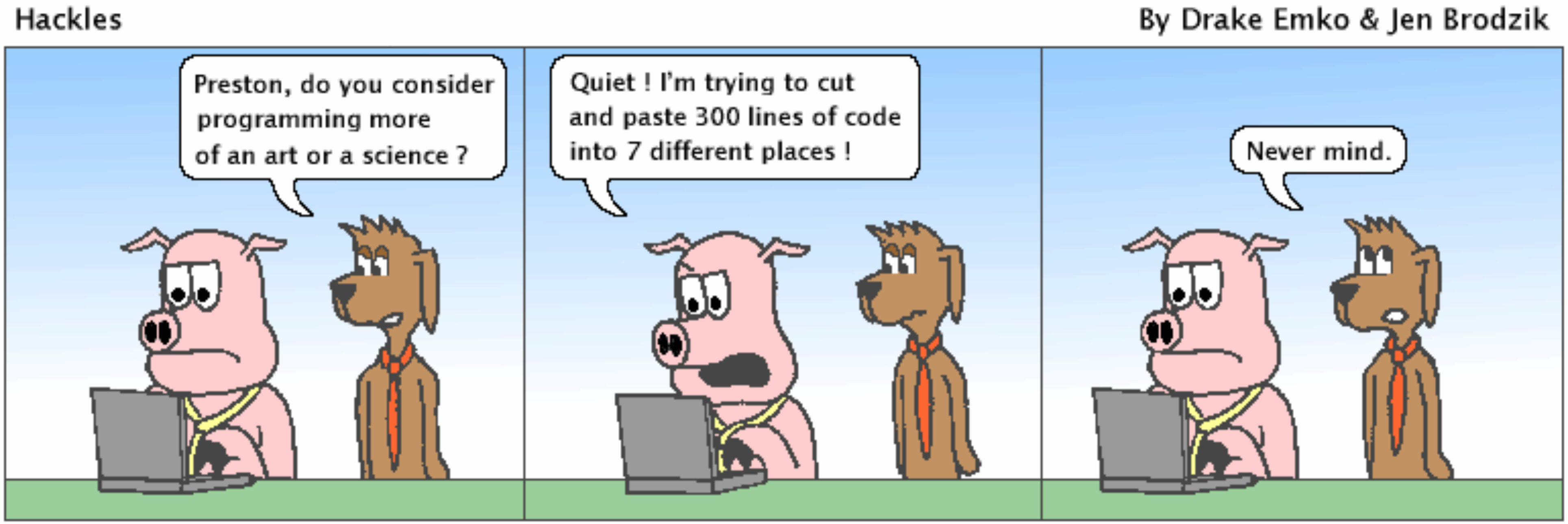
The Set{ } Methods

Function	Alternate Syntax	Description
a.add(x)	N/A	Add element x to the set a
a.remove(x)	N/A	Remove element x from the set a
a.union(b)	a|b	All unique elements in a and b.
a.intersection(b)	a&b	All elements in both a and b.
a.difference(b)	a-b	The elements in but not in b.
a.symmetric_difference(b)	a^b	All elements in a or b but not both.
a.issubset(b)	N/A	True if all elements of a are in b.
a.issuperset(b)	N/A	True if all elements of b are in a.
a.isdisjoint(b)	N/A	True if a and b have no elements in common.

Up next …

Code Organisation

- Functions
  - □ Namespaces
  - □ Scope
  - □ Local functions
  - □ Lambda functions
  - □ Currying
  - □ A closer look at functions (parameters vs arguments)


Hackles
By Drake Emko & Jen Brodzik
Preston, do you consider programming more of an art or a science ?
Quiet ! I'm trying to cut and paste 300 lines of code into 7 different places !
Never mind.

Functions

- Some problems can be very complex. In such cases, it may be easier to separate the problem into smaller, simpler tasks.

- This process is called decomposition.

- Alternatively, sometimes a calculation is very popular (e.g. calculating the mean) and is repeated often and in different parts of a program.

- We don't want to rewrite the code each time it is needed!

- In these situations, we can use functions to help us out.

- Functions are the primary and most important method of code organisation and re-use in Python.

Advantages of Functions

- Program development more manageable

  The divide-and-conquer approach makes program development more manageable. Construct program from smaller pieces/modules.

- Simpler, streamlined code

  Typically simpler and easier to understand when code is broken down into functions.

- Software reusability

  Use existing functions as building-blocks to create new programs.

- Avoid repeating code

  Reduce the duplication of code within a program.

- Better testing

  Testing and debugging becomes simpler when each task within a program is contained in its own function. Test each function in a program individually to determine whether it correctly performs its operation. Easier to isolate and fix errors.

Functions

- A self-contained segment of code which implements a specific, well-defined task.
- Python programs typically combine user-defined functions with library functions available in the standard library.
- A function is invoked by a call which specifies:
  - □ A function name
  - □ Possibly parameters/arguments.
- A function may also return a value.
- `input` and `print` are examples of standard library functions and are invoked in the following way:

```
print("Welcome to functions!")
num = input("Please enter number: ")
```

Function Definition

- Functions are declared using the `def` keyword and are exited using the `return` keyword:

```
def functionName(parameters):
    function-commands
    return expression
```

- The keyword `def` introduces a function definition.
- The `functionName` follows the same rules as variable names.
- The `parameters` are identifier names that will be used as the names *within the function* for the objects passed to the function.
- Parameters are optional. if there is more than one, they are written as a sequence of comma-separated identifiers.

Function Definition

```
def functionName(parameters):
       function-commands
       return expression
```

- function_commands is a sequence of Python statements. These commands must be indented.
- function_commands contain definitions and statements.
- A return statement without an expression argument returns None.
- If no return statement is present (i.e. falling off the end of a function), the function also returns None.
- All functions return a value, even if it is the special value None.

Example: Functions

- Write a function which will take two numbers as parameters, sum them, and return the result.

```
def addNumbers(num1, num2):
    total = num1 + num2
    return total
```

- We can call it from the command prompt as follows:

```
>>> ans = addNumbers(4,5)
>>> print(ans)
9
```

or

```
>>> print(addNumbers(4,5))
```

Namespace, Scope, Local Functions

- Functions can access variables in two different scopes: global and local.
- Namespace: the name we use to describe a variable scope in Python.
- Variables assigned within a function are assigned to the local namespace.
- Local namespace: created when the function is called and immediately populated by the function's arguments. After the function is finished, the local namespace is destroyed.
- Once the function has finished execution, we no longer have access to the variables inside of the function.
- Variables that have global scope can be accessed anywhere within the Python code – useful, but constitute bad programming style!
- Use sparingly!

Example: namespace and scope

- Find the maximum of three numbers.

```
def maximum(n1, n2, n3):
    maxValue = n1
    if (n2 > maxValue):
        maxValue = n2
    if (n3 > maxValue):
        maxValue = n3

    return maxValue
```

```
>>> maxnum = maximum(4,7,1)
>>> print('The maximum is: ',maxnum)
7

>>> print('maxValue is : ',maxValue)

Traceback (most recent call last):
  File "<pyshell#618>", line 1, in <module>
    print('maxValue is: ',maxValue)
NameError: name 'maxValue' is not defined
```

Generates an error as we try to access maxValue which only has a local namespace inside of `maximum`

Returning More Than One Value

- By definition across programming languages, a function is designed to return a single value or object.

- The same is true in Python. However we can get around this limitation by returning a container type, such as a list, e.g.:

```
def example_function():
    return [1, 2, 3]          # packing the values into a tuple

>>> a,b,c = example_function()          # unpack the tuple
>>> a
1
>>> b
2

>>> c
3
```

Returning More Than One Value

- Since we can return an object, we are not restricted to just lists – e.g. we can return a `dict{}` structure instead:

```
def example_function():
    return {'a' : 5, 'b' : 6, 'c' : 7}

>>> dt = example_function()  # packing the values into a dict

>>> dt.keys()                        # Look at the keys
['a', 'c', 'b']

>>> dt.values()                      # Check the values
[5, 7, 6]

>>> dt['a']
5
```

Lambda Functions

- Python supports anonymous or lambda functions.
- These are simple functions consisting of a single statement, the result of which is the return value.
- They are defined using the lambda keyword. This signals to Python that we are declaring an anonymous function. For example:

Using a function

```
>>> def by_2(x):
        return x*2


>>> by_2(3)
6
```

lambda function equivalent

```
>>> by_2 = lambda x: x*2



>>> by_2 (3)
6
```

The lambda function allows us to avoid excess overhead and will be useful in data analysis

Another Lambda Function Example

```
>>> def apply_to_list(some_list, f):
        return [f(x) for x in some_list]


>>> ints = [4, 0, 1, 5, 6]
>>> apply_to_list(ints, lambda x: x * 2)
[8, 0, 2, 10, 12]
```

- Lambda functions can be very powerful and allow for quick and easy manipulation of data.

- Getting your head around generic functions might take some practice, but you'll get there!

Currying

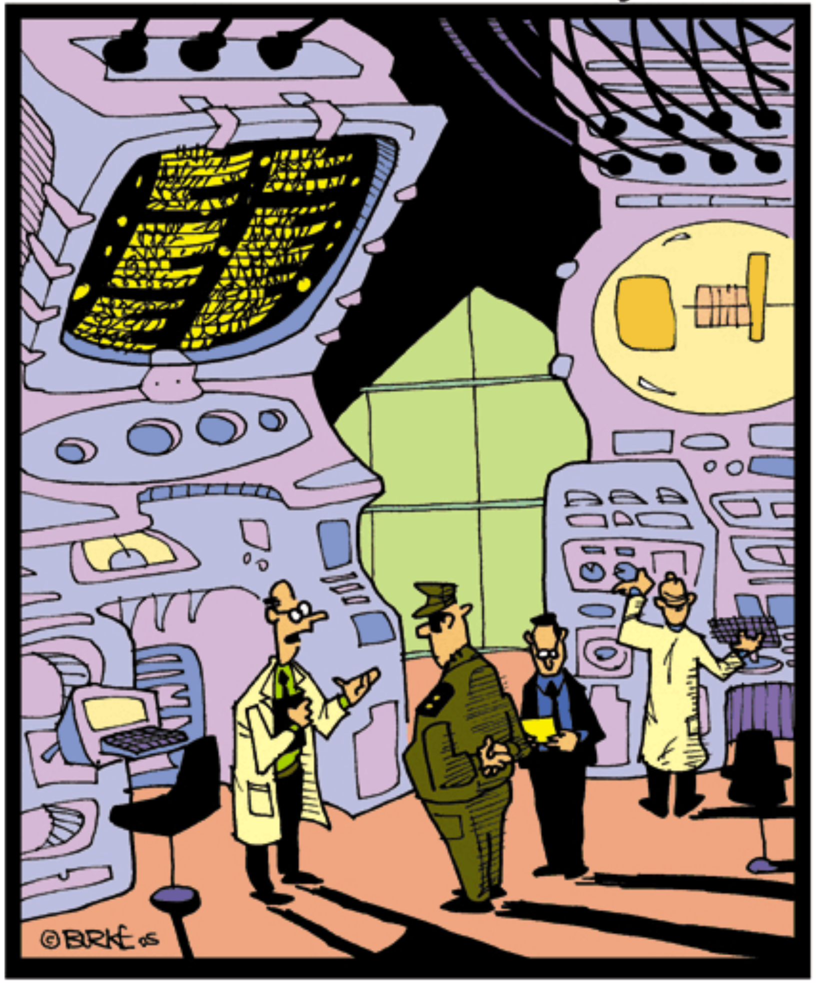
- Currying: the term used when we derive new functions from existing ones by partial argument application. There are two ways we can achieve this:

```
>>> def add_numbers(x, y):
        return x + y

>>> # Lambda function definition of add_five
>>> add_five = lambda y: add_numbers(5, y)
>>> add_five(10)
15

>>> # Alternative definition of add_five
>>> from functools import partial
>>> add_five = partial(add_numbers,5)
>>> add_five(10)
15
```

- This is particularly useful when using `pandas` and transforming time series data.

A closer look at functions …

BEAR FACTS by Burke

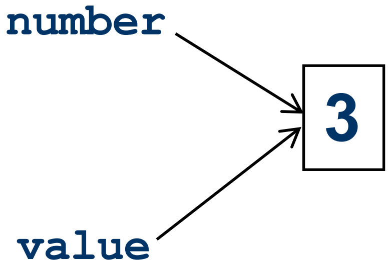


"It may take another week or two for the supercomputers to figure out the best mobile phone plan ..."

Function Arguments and Parameters

- When we invoke a function, we need to specify a function name and we may also specify arguments (or parameters).
- Argument: any piece of data passed into a function when the function is called.
  - □ A function can use its arguments in calculations or other operations.
  - □ When calling the function, the argument is placed in parentheses following the function name.
- Parameter: is a variable that receives an argument passed into a function.
  - □ A variable assigned the value of an argument when the function is called.
  - □ The parameter and the argument reference the same value.

```
def example_function(number):
    result = number * 2
    print(result)

value = 3
example_function(value)
```

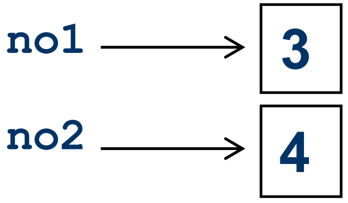


Which one is the argument?

Function Arguments and Parameters

- As we have seen, a function may accept multiple arguments.
- Parameter list items are separated by a comma.
- Arguments are passed by position to the corresponding parameters.
- For this reason arguments must be passed in the exact order in which they are defined as they are positional arguments.

```
def example_function(no1, no2):
    print(no1, no2)

example_function(3, 4)
```



no1 → 3

no2 → 4

There is however an exception …

Keyword Arguments

- Keyword arguments: allow out-of-order parameters.

- Caveat: you must provide the name of the parameter as a 'keyword' to have your arguments match up to their corresponding argument names.

- Example:

```
>>> def example_function(no1, no2):
        print(no1, no2)


>>> example_function(no2=3, no1=4)
4 3
```

Default Arguments

- Default arguments: arguments declared with default values.
- Parameters are defined to have a default value if one is not provided in the function call for that argument.
- Definitions are provided in the function declaration header.
- Positional arguments must come before any default arguments.

```
>>> def example_function(no1, no2=7, no3=2):
  print(no1,no2,no3)
```

positional argument

default arguments

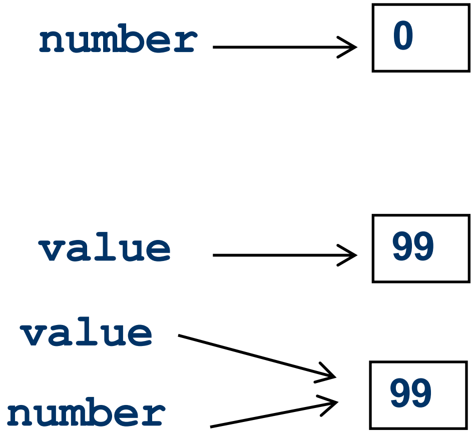
```
>>> example_function(1)
1 7 2

>>> example_function(1,5)
1 5 2
```

Function Arguments and Parameters

- Changes made to a parameter value within a function do not affect the argument.

- For example, the `value` variable passed to the `change_me` function cannot be changed by the function.

```
def change_me(number):
    print('Attempting to change the value!')
    number = 0
    print('Now the value is:', number)

value = 99
print('The value is:', value)
change_me(value)
print('After the function call the value is still:', value)
```



Why does this happen?

Function Arguments and Parameters

- Functions can change the values of arguments if they are:
  - □ mutable; and
  - □ non-primitive values (pass by reference).

```
>>> def change_me(aList, aNum):
        aList.append(aNum)                    #change the list (mutable)
        aNum += 1                             #change the variable?

>>> numList = [1, 2]
>>> num = 3
>>> print(num, numList)                   # Output is: 3 [1, 2]

>>> change_me(numList, num)
>>> print(num, numList)                   # Output is: 3 [1, 2, 3]

>>> change_me(numList, num)
>>> print(num, numList)                   # Output is: 3 [1, 2, 3, 3]
```

A closer look at scope

- The part of a program in which a variable may be accessed.
- Variables either have local or global scope.
- Variables defined within a function have local scope.
  - Live as long as the functions they are defined in are active and are accessible within the function only.
- Variables defined at the highest level in a module/file have global scope.
  - Lifespan lasts as long as the program is executing. The values of global variables are accessible to all functions.
- Example:

```
>>> def example_function():
    local_str = 'local'
    print(global_str + ' ' + local_str)

>>> global_str = 'global'
>>> example_function()
global local
```

A closer look at scope

- Variables can be defined:
  - □ At the top level in a program.
  - □ In a block.

- If you define a variable in a block:
  - □ It is distinct from any variable of the same name defined outside the block.
  - □ When you leave the block, you no longer have access to the variable.

- Parameters to functions are treated as variables defined in the block of the function.

- When searching for an identifier/variable, Python searches the local scope first, then the global scope.

Global Variables

- Global variable names can be overridden by local variables, e.g.:

```
>>> def example_function():
        global_str = 'local'
        print('String in function is:', global_str)

>>> global_str = 'global'
>>> print('String is:', global_str)
String is: global

>>> example_function()
String in function is: local

>>> print('String is:', global_str)
String is: global
```

Global Variables inside a Function

- We can also assign a value to a global variable inside of a function by declaring the global variable.
- We use the word **`global`** to ensure the global version of global_str is used.
- To reference/access a global variable, use the global statement, e.g.:

```
>>> def example_function():
        global global_str
        global_str = 'local'
        print('String in function is:', global_str)

>>> global_str = 'global'
String is: global

>>> print('String is:', global_str)
>>> example_function()
 String in function is: local

 >>> print('String is:', global_str)
String is: local
```

Global Variables and Constants

- Most programmers agree that you should restrict the use of global variables, or not use them at all. A few reasons why:
  - □ Make debugging difficult: many locations in the code could be causing a wrong variable value.
  - □ Functions that use global variables are usually dependent on those variables. Harder to transfer functions to another program.
  - □ Make a program hard to understand.
- Preferred approach: create variables locally and pass them as arguments to the functions as needed.
- Only use the information passed to functions via the parameters or defined within the function.

Global Variables and Constants

- Global constants however, are permissible to use in a program.

- Global constant: a global name that refers to a value that cannot be changed.

- Although Python does not allow you to create true global constants, you can simulate them with global variables.
  - □ For example:

```
MAX_NUMBER = 10
```

- It is common practice to write a constant's name in uppercase letters.

- Reminds us that the value referenced by the name is not to be changed in the program!

Scope Exercise: What is the output of the code?

```
j = 1
k = 2
def function1():
    j = 3
    k = 4
    print('j is:', j, 'k is', k)

def function2():
    j = 6
    function1()
    print('j is:', j, 'k is', k)


k = 7
function1()
print('j is:', j, 'k is', k)


j = 8
function2()
print('j is:', j, 'k is', k)
```

Try to work out the output from this code, WITHOUT entering the code into Python first.

You can check yourself afterwards by implementing and running the code.