J. Hageman
Hanze *University of Applied Sciences*
Groningen

Bioinformatics data processing and –analysis using the scripting language Python

Introduction to programming with Python

Data types and collections

Programs and data

- All programs need data to work on
- To store data, you need a placeholder for it: these are called ***variables***. Here are a few examples:

```
1    amino_acid = 'alanine'
2    number_of_atoms = 13
3    mw = 89.09
4
```

Creating variables

- Here you can find a movie that explains variables.
- Information (data) comes in many forms, such as numbers, characters, words, pictures, sound. In programming languages like python information is stored in variables. You can think of a box that has a label on it and stuff is stored into that. You can find or use the information later by searching for the box with the label you stored the information in.

Creating variables

- What happens when I type **`amino_acid = 'alanine'`** ?

amino_acid is the *name* of the *variable* you have just created

The *assignment operator* **=** was used to assign the string alanine to the variable amino_acid

'alanine' is a *literal*, the *value* to be assigned to the variable amino_acid

Introduction

Python has several data types. We will cover several types.

- None
- Booleans: True/False
- Strings: “Hello”
- Integers: 7
- Floats: 2.5

Introduction

Python has several collections. We will cover several collections.

- List: mutable, ordered collection
- Tuple: immutable, ordered collection
- Set: unordered collection of unique items
- Dictionary: unordered library of key-value pairs

None

- None is just a value that commonly is used to signify 'empty', or 'no value here'. It is a *signal object.*
- This way, you can declare a variable with empty data, preventing a NameError.

```
>>> data
Traceback (most recent call last):
  File "<pyshell#77>", line 1, in <module>
    data
NameError: name 'data' is not defined
>>> data = None
>>> data
>>>
```

Booleans

There are only two possible booleans:

- True
- False

- In Python you can convert many datatypes to boolean with the bool() function

Booleans

```
>>> num = 1
>>> bool(num)
True
>>> num = 0
>>> bool(num)
False
>>> True == 1
True
>>> False == 0
True
>>>
```

The == means a equality test. Python evaluates if the two are equal.

Not only is 1 converted to True. 1 equals True directly! 0 equals False. 1 and True are the same thing in Python!

Booleans

```
>>> my_list = []
>>> bool(my_list)
False
>>> my_list = ["kwek", "kwek", "kwak"]
>>> bool(my_list)
True
>>> str1 = ""
>>> bool(str1)
False
>>> str2 = "Hello"
>>> bool(str2)
True
>>>
```

Empty collections evaluate as False when converted to bool. Non-empy as True. Later more about this.

This holds for strings as well!

Integers

- Working with integers is easy in Python. Just type the number and your done. You can also store integers as variables and do simple math.

```
>>> x = 2
>>> y = 4
>>> z = x + y
>>> z
6
>>>
```

Floats

- Floats represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10

Floats

```
>>> x = 1.6
>>> y = 2
>>> z = x + y
>>> z
3.6
>>> i_am_big = 12e55
>>> i_am_big
1.2e+56
>>>
```

Strings

- Strings are ordered sequences of Unicode characters.
- Strings can be created using single quotes or double quotes.
- Strings are iterables which means that it can take sequential indexes starting from zero.
- Strings have a zero-based index in Python

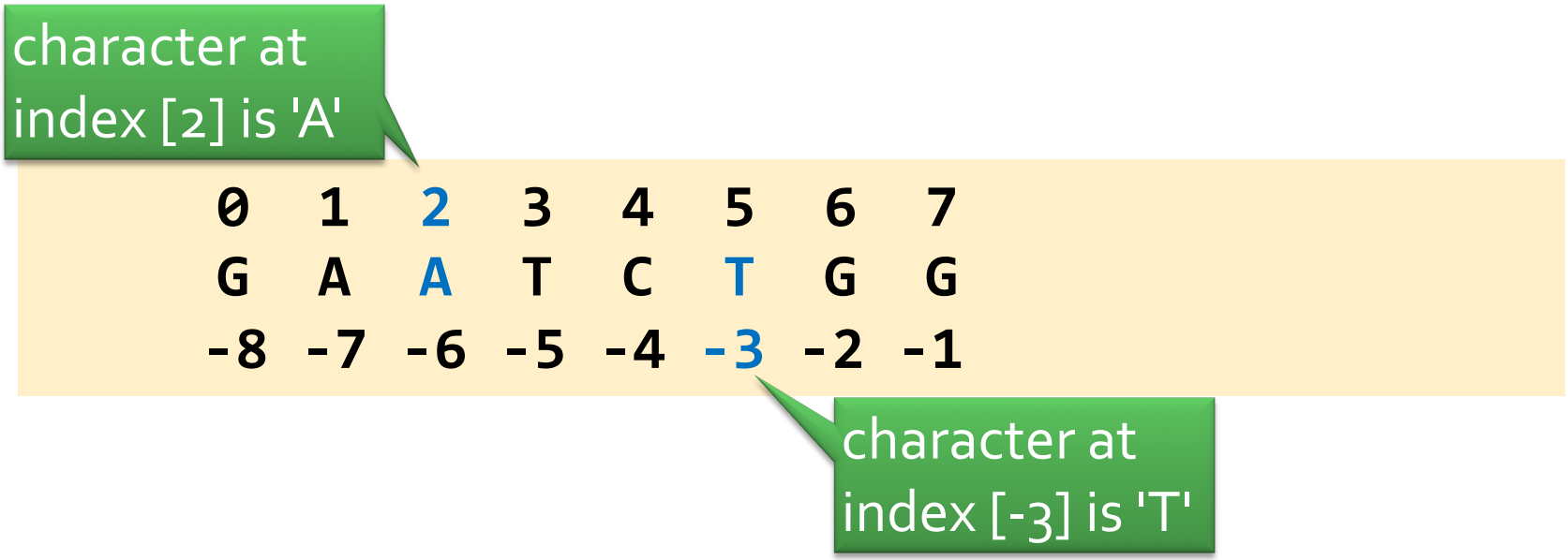
Strings

```
>>> name = "Jan"
>>> #You can also create a string using single quotes
>>> name2 = 'Pien'
>>> #Use of single quote in string:
>>> message = "Let's code some Python code"
>>> #If you want to use single quotes in a string generated by single quotes
>>> #You can use an escape:
>>> message = 'Let\'s code some Python code'
>>>
```

The \ character serves as an escape character!

String indexing

- String positions can be specified using an *index*.
- When working form the start, indexes start at 0
- When working from the end, indexes start at -1



String manipulation

```
>>> seq1 = "GAATTC"
>>> seq2 = "GGATCC"
>>> #String concatenation:
>>> seq1 + seq2
'GAATTCGGATCC'
>>> #String multiplication:
>>> seq1 * 3
'GAATTCGAATTCGAATTC'
>>>
```

String slicing

```
>>> seq1 = "GATC"
>>> #String slicing:
>>> seq1[0]
'G'
>>> seq1[1]
'A'
>>> seq1[-1]
'C'
>>> seq1[-2]
'T'
>>>
```

String slicing

Slicing rules:

- **seq[x:y:z]** will return a slice starting at, and including, index **x** (or o if omitted) to, but excluding, index **y** (or the end if omitted), and only taking the character at each step **z**
- A slice will always return the same data type

```
>>> seq = 'GATC'
>>> #From pos 1 to end (and including end):
>>> seq[1:]
'ATC'
>>> #From pos 0 to 3 (not including 3):
>>> seq[:3]
'GAT'
>>> #Start to end (including end) with step 2:
>>> seq[::2]
'GT'
>>> #Reverse a string:
>>> seq[::-1]
'CTAG'
>>>
```

Wrong string indexing

- What happens when you try to get a position that does not exist?

```
>>> seq = 'GATC'
>>> seq[5]
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    seq[5]
IndexError: string index out of range
>>>
```

If you try to access an index that does not exist, you get an IndexError!

A first glance at objects & methods

- Since Python is a real object-oriented programming language, all data in Python are represented as ***objects***
- In terms of programming, an object is a structure with ***data*** (properties or attributes) and ***methods*** (functionality)
- We will explore these concepts a little with string objects

What is a method?

- A method is a piece of code that is accessible through its name.
- You can *call* a method and cause its code to be executed
- Method calling comes in two flavors:

```
method( arguments )
```

```
object.method( arguments )
```

Method call, flavor 1

- This flavor is used for methods calls within the same script and many build-in methods, such as `str()`

```
method( arguments )
```

method is the name of the piece of code you want to be executed (*e.g.* `find`)

arguments are one or more pieces of data you can pass to a method *e.g.* `find('ATC')`. Some methods do not take any arguments.

Method call, flavor 2

- This flavor is used for methods calls on objects

```
object.method( arguments )
```

object is the piece of data you are working on (*e.g.* a string object)

You use the dot operator ('.') to call a method of an object

A first glance at objects & methods

- Use dir(str) to explore the properties and methods of a string object.

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq_
_', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__
gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__
', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitali
ze', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'for
mat', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidenti
fier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join
', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex
', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', '
strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>>
```

String methods

- String methods that you need to know off:

method	what to use it for
*str.*startswith*('substring')* *str.*endswith*('substring')*	tests whether a given string starts/ends with substring
*str.*strip()	strip off any leading and trailing characters
str.rstrip() *str*.lstrip()	strip off right or left trailing characters
*str.*split('pattern')	split the string in a list of strings on all occurrences of *pattern*
*str.*upper() *str.*lower()	convert string to upper/lower case
str.find()	returns the start position of substring
str.index()	returns the start position of substring

Use of string methods: strip()

```
>>> str1 = "GATC"
>>> str1.startswith('G')
True
>>> str1.endswith('G')
False
>>> str2 = '    a lot of trailing whitespace     '
>>> str2.strip()
'a lot of trailing whitespace'
>>> str3 = 'some typozzzz'
>>> str3.strip('z')
```

Without argument strip will strip whitespace.

Note that strip will remove all repeats of the trailing character

Use of string methods: split()

```
>>> students = "Lisa Jaap Jan Truus"
>>> students.split()
['Lisa', 'Jaap', 'Jan', 'Truus']
>>> dna = 'GATC'
>>> dna.split('T')
['GA', 'C']
>>>
```

Without argument split will split on whitespace.

Note that split will not include the argument in a list

Use of string methods: upper() and lower()

```
>>> dna = 'gatc'
>>> dna2 = "AATT"
>>> dna.upper()
'GATC'
>>> dna2.lower()
'aatt'
>>>
```

Upper and lower will convert the whole string. It does not accept an argument.

Use of string methods: find() and index()

```
>>> dna = 'GATC'
>>> dna.find('A')
1
>>> dna.index('A')
1
>>> dna.find('Q')
-1
>>> dna.index('Q')
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    dna.index('Q')
ValueError: substring not found
>>>
```

When a substring is not found, find will return -1 while index will raise an error!

Warning! Strings are immutable

- If you try to change a character within a string, you will get an error

```
>>> dna = 'gatc'
>>> dna[1] = 't'
Traceback (most recent call last):
  File "<pyshell#45>", line 1, in <module>
    dna[1] = 't'
TypeError: 'str' object does not support item assignment
>>>
```

- The solution: use a **List**

Lists

- Lists are … lists (of objects)
- They are in other languages also called sequences, vectors or arrays
- They contain ordered series of objects: characters, strings, numbers, or any other Python object
- They are quite similar to strings in many ways but with one important exception: they are **mutable**

List basics

- Lists are created using square brackets, with elements separated by commas

```
>>> empty_list = []
>>> sequences = ['atc', 'agg', 'ttt']
>>> sequences[1]
'agg'
>>> sequences[-2]
'agg'
>>> sequences[1:]
['agg', 'ttt']
>>>
```

[] creates an empty list

List indexing works the same as with strings. If you access a single element you get its value; if you access more, you get a new list

List basics: extend

```
>>> sequences = []
>>> sequences.append("atc")
>>> sequences
['atc']
>>> sequences = sequences + ["atg"]
>>> sequences
['atc', 'atg']
```

Append and concatenating lists can both be used to extend

List: change a list

```
>>> sequences = ['atg', 'caa']
>>> #prepend list
>>> sequences = ['aaa'] + sequences
>>> sequences
['aaa', 'atg', 'caa']
>>> #replace item
>>> sequences[1] = 'ggg'
>>> sequences
['aaa', 'ggg', 'caa']
>>> #remove item
>>> sequences.remove('ggg')
>>> sequences
['aaa', 'caa']
>>>
```

List: delete and insert multiple items

```
>>> sequences = ['atg', 'caa']
>>> #insert multiple items:
>>> sequences[1:1] = ['ttt', 'ggg']
>>> sequences
['atg', 'ttt', 'ggg', 'caa']
>>> #delete multiple items:
>>> sequences[1:3] = []
>>> sequences
['atg', 'caa']
>>>
```

Nested Lists

- You can also create lists within lists: *nested* lists or *matrices (also called multi-dimensional lists)*

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

this is a list of three lists of two elements each

and this returns the second element of the second row

1	2	3
4	**5**	6
7	8	9

Some useful list methods

```
>>> list1 = [1, 2, 3, 3, 4, 5]
>>> list1.index(4)
4
>>> list1.count(3)
2
>>>
```

list.index(x)
returns the index of the first occurrence of x in list

list.count(x)
returns the number of occurrences of x in list

Some useful functions

- These functions are useful on lists but work on any iterable

```
>>> list1 = [4, 6, 3, 7, 9]
>>> min(list1)
3
>>> max(list1)
9
>>> len(list1)
5
>>> sum(list1)
29
>>>
```

These functions are self-explanatory

Warning! copy of list

- Frequent cause of code errors
- No need to understand this thoroughly but be aware that lists (as any object) references are passed by value. The following code snippet will explain:

```
>>> list1 = [0]
>>> list1
[0]
>>> list2 = list1
>>> list2.append(1)
>>> list2
[0, 1]
>>> list1
[0, 1]
>>> list1 is list2
True
>>>
```

list1 and list2 both refer to the same object.
list2 is NOT a copy of list1! They are the same thing!

Solution! copy list

- If you want a copy of a list do NOT assign it to a new variable but use [:] instead:

```
>>> list1 = [0]
>>> list1
[0]
>>> list2 = list1[:]
>>> list2
[0]
>>> list2.append(1)
>>> list2
[0, 1]
>>> list1
[0]
>>> list1 is list2
False
>>>
```

The slice [:] copies the objects of the list from beginning to end including the end.

list1 and list2 do not point to the same object in memory

Tuples

- Tuples are like lists, but are as **immutable** as strings
- They contain ordered series of data elements: characters, strings, numbers or any other Python object
- They share a lot of methods and operations with strings and lists

Tuples

- Tuples are created using parentheses, with elements separated by commas

```
>>> dna = ('atc', 'atg', 'ttt')
>>> dna[1]
'atg'
>>> dna = dna + ('aaa')
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    dna = dna + ('aaa')
TypeError: can only concatenate tuple (not "str") to tuple
>>>
```

tuples are indexed just like lists and strings

tuples, like strings, cannot be changed

Why use tuples?

- Tuples make code more safe. Because they are immutable you will be sure they are not changed
- Tuples can be used as a key in a dictionary (see below). This is not possible with lists.

```
>>> dict1 = {("adenine","thymine"):"pyrimidine"}
>>> dict1[("adenine", "thymine")]
'pyrimidine'
>>> dict2 = {["adenine","thymine"]:"pyrimidine"}
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    dict2 = {["adenine","thymine"]:"pyrimidine"}
TypeError: unhashable type: 'list'
>>>
```

Shared sequence operations: list, string, tuple

Operation	Meaning
`x in s`	True if an item of s is equal to x, else False
`x not in s`	False if an item of s is equal to x, else True
`s + t`	the concatenation of s and t
`s * n, n * s`	n shallow copies of s concatenated
`s[i]`	i'th item of s, origin o
`s[i:j]`	slice of s from i to j
`s[i:j:k]`	slice of s from i to j with step k
`len(s)`	length of s
`min(s)`	smallest item of s
`max(s)`	largest item of s
`s.index(i)`	index of the first occurence of i in s
`s.count(i)`	total number of occurences of i in s

Dictionaries

- Sometimes, you want fast access of values by some key; a simple list of data will not do
- Suppose you want to store the amino acid translations for all 64 codons, where an amino acid is returned for a given codon (*e.g.* "GGG" will return "Gly"
- The dictionary is the ideal structure to store this kind of data

Dictionaries: keys and values

- Here is how you could store the universal codon table in Python

```
>>> codon_table = { 'AAA':'K', 'CCC':'P', 'GGG':'G', 'TTT':'F'}
>>> codon_table['CCC']
'P'
>>> codon_table['ZZZ']
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    codon_table['ZZZ']
KeyError: 'ZZZ'
>>>
```

create a dictionary with four entries

get a value based on a key

error if you try to access a key that does not exists

Dictionaries: keys and values

```
>>> codon_table = { 'AAA':'K', 'CCC':'P', 'GGG':'G', 'TTT':'F'}
>>> codon_table['GGC'] = 'G'
>>> codon_table
{'TTT': 'F', 'AAA': 'K', 'CCC': 'P', 'GGC': 'G', 'GGG': 'G'}
>>> codon_table.update({'GGC':'G'})
>>> codon_table
{'GGC': 'G', 'AAA': 'K', 'TTT': 'F', 'CCC': 'P', 'GGG': 'G'}
>>> del(codon_table['GGC'])
>>> codon_table
{'AAA': 'K', 'TTT': 'F', 'CCC': 'P', 'GGG': 'G'}
>>>
```

create a new entry

alternative to create a new entry

delete an entry

Dictionaries: prevent error if key does not exist

```
>>> codon_table
{'AAA': 'K', 'TTT': 'F', 'CCC': 'P', 'GGG': 'G'}
>>> codon_table['TTT']
'F'
>>> codon_table['QQQ']
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    codon_table['QQQ']
KeyError: 'QQQ'
>>> codon_table.get('TTT')
'F'
>>> codon_table.get('QQQ')
>>>
```

The dict.get() method returns None if a key does not exist.

Dictionaries are unordered

- Dictionaries are, in contrast to lists and tuples, **unordered**.
- That means the elements don't necessarily come out the way they are put into it

Python Arithmetic Operators

Expression	Description
x + y	addition
x - y	subtraction
x * y	multiplication
x / y	division
x % y	modulus
x ** y	exponent
x // y	floor division

https://www.tutorialspoint.com/python/python_basic_operators.htm

Python Arithmetic Operators: examples

```
>>> x = 3
>>> y = 2
>>> x + y
5
>>> x - y
1
>>> x * y
6
>>> x / y
1.5
>>> x % y
1
>>> x // y
1
>>>
```

3 / 2 = 1, remainder = 1
=> thus 3 % 2 = 1

3 / 2 = 1.5 , chop of decimal.
=> thus 3 // 2 = 1

Python Comparison Operators

Expression	Description
x == y	x equals y
x < y	x is less than y
x > y	x is greater than y
x >= y	x is greater than or equal to y
x <= y	x is less than or equal to y
x != y	x is not equal to y

https://www.tutorialspoint.com/python/python_basic_operators.htm

Python Comparison Operators: examples

```
>>> x = 3
>>> y = 2
>>> x == y
False
>>> x < y
False
>>> x > y
True
>>> y = 3
>>> x >= y
True
>>> x <= y
True
>>> x != y
False
>>>
```

y is now similar to x!

Python assignment operators

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+=	adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-=	subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*=	multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%=	takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**=	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//=	performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

https://www.tutorialspoint.com/python/python_basic_operators.htm

Python assignment operators: examples

```
>>> x = 2
>>> x += 1
>>> x
3
>>> x *= 2
>>> x
6
>>> x **= 2
>>> x
36
>>>
```

Python membership and identity operators

Operator	Description
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.

https://www.tutorialspoint.com/python/python_basic_operators.htm

Python membership and identity operators: examples

```
>>> list1 = [1, 2, 3]
>>> list2 = [1, 2, 3]
>>> 2 in list1
True
>>> 2 not in list1
False
>>> list1 == list2
True
>>> list1 is list2
False
>>> id(list1)
4434285128
>>> id(list2)
4300914504
>>>
```

list1 has identical content as list 2

But they are not the same object!

This shows that the list objects have different identities

The Math module

- The math module has some neat math-related properties and methods
- Simply import it with 'import math'
- Use dir(math) to explore all the properties and methods

The Math module

Math property/method	Description
math.pi	he mathematical constant π = 3.141592..., to available precision.
math.e	The mathematical constant e = 2.718281..., to available precision.
math.log10(*x*)	Return the base-10 logarithm of *x*
math.log(*x*[, *base]*)	With one argument, return the natural logarithm of *x* (to base *e*). With two arguments, return the logarithm of *x* to the given *base*.
math.pow(*x*, *y*)	Return x raised to the power y.
math.exp(*x*)	Return e**x.

The Math module

Math property/method	Description
math.sin(x) math.cos(x) math.tan(x)	Return the sine, cosine or tangent of *x* radians. Note it expects radians, not degrees!
`math.sqrt(x)`	Return the square root of *x*.
`math.ceil(x)` `math.floor(x)`	Return the ceiling or floor of *x*, the smallest integer greater than or equal to *x*

The Math module: examples

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>> math.sqrt(16)
4.0
>>> math.sin(0.5*math.pi)
1.0
>>> math.log(100, 10)
2.0
>>> math.ceil(2.6)
3
>>>
```

The random module

- To use (pseudo) random numbers, use the random module
- Simply type: import random
- Note that true random numbers are difficult to achieve for computers
- dir(random) will give you all methods and properties.
- random.randint() and random.choice() are the most interesting for now

The random module: examples

```
>>> import random
>>> #generate random number from 1-10, note that this includes 10
>>> random.randint(1, 10)
4
>>> list1 = ['python', 'java', 'C']
>>> #get random item from list
>>> random.choice(list1)
'C'
>>>
```