 Computers have their own language called machine language, which is very rudimentary.

an alphabet

a set of symbols used to build words of a certain language

a lexis

(aka a dictionary) a set of words the language offers its users

a syntax

a set of rules (formal or informal, written or felt intuitively) used to determine if a certain string of words forms a valid sentence

semantics

a set of rules determining if a certain phrase makes sense

high-level programming languages:a language in which humans can write their programs and a language that computers may use to execute the programs, one that is far more complex than machine language and yet far simpler than natural language.

A complete set of known commands is called an instruction list,The IL is, in fact, the alphabet of a machine language

A program written in a high-level programming language is called a source code

There are two different ways of transforming a program from a high-level programming language into machine language:

**Compilation**: the source code is translated once (it has to be recompiled any time the code is modified)by getting a file with the machine code,you can distribute the file without the ***compiler/translator***(the program that converts the source code to machine code)

**Interpretation**: here, the source code is translated any time it has to be run. However, the source code has to be distribute with the ***interprete***r(a it is used every time the code is run).

The compiler translates (compiles) the source program (a set of high-level language instructions) into machine code (a lower-level representation of the source program). The sequence of 0s and 1s can be then executed by the processor. The execution of the compiled code is usually faster than interpretation.

A particular high-level programming language is designed to fall into one of these two categories

languages designed to be utilized in the interpretation manner are often called scripting languages, while the source programs encoded using them are called scripts.

Python is a widely-used, interpreted, object-oriented, and high-level programming language with dynamic semantics, used for general-purpose programming.the name of the Python programming language comes from an old BBC television comedy sketch series called Monty Python's Flying Circus.

Python was created by ***[Guido van Rossum](https://en.wikipedia.org/wiki/Guido_van_Rossum" \t "https://www.netacad.com/content/pe1/1.0/index.html?xAPILaunchKey=a867e461-7b4e-43e0-b782-4f5c6f1ed1d6&xAPILaunchService=https://www.netacad.com/adl/content/&lang=en-US&moduleNumber=1" \l "/courses/content/m1/id/_blank)***, born in 1956 in Haarlem, the Netherlands.

There are two main kinds of Python, called Python 2 and Python 3. These two versions of Python ***are not*** compatible with each other

Python 2 is an older version of the original Python. Its development has since been intentionally stalled,but updates are issued on a regular basis, not intended to modify the language in any significant way, but rather fix any freshly discovered bugs and security holes. Python 2's development path has reached a dead end already, but Python 2 itself is still very much alive.

Python 3 is the current version of the language. it is a completely different language, although it's very similar to its predecessor.  There are too many existing Python 2 applications to discard it altogether.This is the reason why Python 2 is still in use.

An implementation of Python refers to "a program or environment, which provides support for the execution of programs written in the Python language, as represented by the CPython reference implementation."The traditional implementation of Python, called CPython, is Guido van Rossum's reference version of the Python computing language.

PYTHON

The word print that you can see here is a function name. It comes from Python itself.The function is an added value received together with Python and its environment (it is built-in).They may come from one or more of Python's add-ons named modules.

Python functions strongly demand the presence of a pair of parentheses. If you're going to use a function which doesn't take any argument, you still have to have the parentheses, to distinguish ordinary words from function names.If you want to deliver one or more arguments to a function, you place them inside the parentheses.

Python's syntax, requires that there cannot be more than one instruction in a line.

the escape character, eg: \ (backslash)

Positional arguments means the arguments are dictated by its position.

Keyword arguments. The name stems from the fact that the meaning of these arguments is taken not from its location (position) but from the special word (keyword) used to identify them,

The print() function has two keyword arguments that you can use for your purposes. The first is called end and second sep.

Any keyword arguments have to be put after the last positional argument (this is very important)

The **print()** function is a ****built-in**** function. It prints/outputs a specified message to the screen/console window.

2. Built-in functions, contrary to user-defined functions, are always available and don't have to be imported. Python 3.8 comes with 69 built-in functions. You can find their full list provided in alphabetical order in the [Python Standard Library](https://docs.python.org/3/library/functions.html" \t "https://www.netacad.com/content/pe1/1.0/index.html?xAPILaunchKey=cede1160-efd7-4120-b656-40c27190586c&xAPILaunchService=https://www.netacad.com/adl/content/&lang=en-US&moduleNumber=2" \l "/courses/content/m2/id/_blank).

3. To call a function (this process is known as ****function invocation**** or ****function call****), you need to use the function name followed by parentheses. You can pass arguments into a function by placing them inside the parentheses. You must separate arguments with a comma, e.g., **print("Hello,", "world!")**. An "empty" **print()** function outputs an empty line to the screen.

4. Python strings are delimited with ****quotes****, e.g., **"I am a string"** (double quotes), or **'I am a string, too'** (single quotes).

5. Computer programs are collections of ****instructions****. An instruction is a command to perform a specific task when executed, e.g., to print a certain message to the screen.

6. In Python strings the ****backslash**** (**\**) is a special character which announces that the next character has a different meaning, e.g., **\n** (the ****newline character****) starts a new output line.

7. ****Positional arguments**** are the ones whose meaning is dictated by their position, e.g., the second argument is outputted after the first, the third is outputted after the second, etc.

8. ****Keyword arguments**** are the ones whose meaning is not dictated by their location, but by a special word (keyword) used to identify them.

9. The **end** and **sep** parameters can be used for formatting the output of the **print()** function. The **sep** parameter specifies the separator between the outputted arguments, e.g., **print("H", "E", "L", "L", "O", sep="-")**, whereas the **end** parameter specifies what to print at the end of the print statement.

2.

A literal is data whose values are determined by the literal itself. The characteristic of the numeric value which determines its kind, range, and application, is called the type.

you can write this number either like this: 11111111, or like this: 11\_111\_111.

Nb: \*Python 3.6 has introduced underscores in numeric literals, allowing for the placement of single underscores between digits and after base specifiers for improved readability.

1. ****Literals**** are notations for representing some fixed values in code. Python has various types of literals - for example, a literal can be a number (numeric literals, e.g., **123**), or a string (string literals, e.g., "I am a literal.").

2. The ****binary system**** is a system of numbers that employs *2* as the base. Therefore, a binary number is made up of 0s and 1s only, e.g., **1010** is *10* in decimal.

Octal and hexadecimal numeration systems, similarly, employ *8* and *16* as their bases respectively. The hexadecimal system uses the decimal numbers and six extra letters.

3. ****Integers**** (or simply ****int****s) are one of the numerical types supported by Python. They are numbers written without a fractional component, e.g., **256**, or **-1** (negative integers).

4. ****Floating-point**** numbers (or simply ****float****s) are another one of the numerical types supported by Python. They are numbers that contain (or are able to contain) a fractional component, e.g., **1.27**.

5. To encode an apostrophe or a quote inside a string, you can either use the escape character, e.g., **'I\'m happy.'**, or open and close the string using an opposite set of symbols to the ones you wish to encode, e.g., **"I'm happy."** to encode an apostrophe, and **'He said "Python", not "typhoon"'** to encode a (double) quote.

6. ****Boolean values**** are the two constant objects **True** and **False** used to represent truth values (in numeric contexts **1** is **True**, while **0** is **False**.

**Extra**

There is one more, special literal that is used in Python: the **None** literal. This literal is a **NoneType** object, and it is used to represent ****the absence of a value****. We'll tell you more about it soon.

## Key takeaways

1. An ****expression**** is a combination of values (or variables, operators, calls to functions ‒ you will learn about them soon) which evaluates to a certain value, e.g., 1 + 2.

2. ****Operators**** are special symbols or keywords which are able to operate on the values and perform (mathematical) operations, e.g., the \* operator multiplies two values: x \* y.

3. Arithmetic operators in Python: + (addition), - (subtraction), \* (multiplication), / (classic division ‒ always returns a float), % (modulus ‒ divides left operand by right operand and returns the remainder of the operation, e.g., 5 % 2 = 1), \*\* (exponentiation ‒ left operand raised to the power of right operand, e.g., 2 \*\* 3 = 2 \* 2 \* 2 = 8), // (floor/integer division ‒ returns a number resulting from division, but rounded down to the nearest whole number, e.g., 3 // 2.0 = 1.0)

4. A ****unary**** operator is an operator with only one operand, e.g., -1, or +3.

5. A ****binary**** operator is an operator with two operands, e.g., 4 + 5, or 12 % 5.

6. Some operators act before others - ****the hierarchy of priorities****:

* the \*\* operator (exponentiation) has the highest priority;
* then the unary + and - (note: a unary operator to the right of the exponentiation operator binds more strongly, for example 4 \*\* -1 equals 0.25)
* then: \*, /, and %,
* and finally, the lowest priority: binary + and -.

7. Subexpressions in ****parentheses**** are always calculated first, e.g., 15 - 1 \* (5 \* (1 + 2)) = 0.

8. The ****exponentiation**** operator uses ****right-sided binding****, e.g., 2 \*\* 2 \*\* 3 = 256.

1. A ****variable**** is a named location reserved to store values in the memory. A variable is created or initialized automatically when you assign a value to it for the first time. (2.1.4.1)
2. Each variable must have a unique name ‒ an ****identifier****. A legal identifier name must be a non-empty sequence of characters, must begin with the underscore(**\_**), or a letter, and it cannot be a Python keyword. The first character may be followed by underscores, letters, and digits. Identifiers in Python are case-sensitive.
3. Python is a ****dynamically-typed**** language, which means you don't need to *declare* variables in it. (2.1.4.3) To assign values to variables, you can use a simple assignment operator in the form of the equal (**=**) sign, i.e., **var = 1**.
4. You can also use ****compound assignment operators**** (shortcut operators) to modify values assigned to variables, for example: **var += 1**, or **var /= 5 \* 2**.
5. You can assign new values to already existing variables using the assignment operator or one of the compound operators, for example:

Nb:

The digraph, called the newline character, causes the current line to end at the point indicated by the digraph, and creates a new line that starts right after it.

Keyword parameters (also called named parameters) are parameters that have values determined by a keyword name followed by an equals sign (=) and a default value assigned to that keyword name

If an integer number is preceded by **0o or 0O**, it will be treated as an **octal value**. For example: 0o246 is an octal number with a decimal value equal to 166.

If an integer number is preceded by**0x or 0X**, it will be treated as a **hexadecimal value**. For example: 0x246 is a hexadecimal number with a decimal value equal to 582.

***Converting 0x1234 to decimal:***

46600x1234 = (1×163) + (2×162) + (3×161) + (4×160)=4660

Finally, if an integer number is preceded by **0b or 0B**, it will be treated as a **binary value**. For example: 0b1111 is a binary number with a decimal value equal to 15.

The / operator is one of the two types of division operator in Python that divides its left operand by its right operand, and returns a floating-point value.

The // operator, called the floor division operator, performs a similar operation, but rounds down the result and returns an integer number.

3.

**=** is an assignment operator, e.g., a = b assigns a with the value of b;

**==** is the question are these values equal? so a == b compares a and b.It is a binary operator with left-sided binding. It needs two arguments and checks if they are equal.

The != (not equal to) operator compares

while conditional\_expression:

instruction\_one

…...    :

    instruction\_n

The semantic difference is more important: when the condition is met, if performs its statements only once; while repeats the execution as long as the condition evaluates to True.

* an instruction or set of instructions executed inside the while loop is called the ****loop's body****;

An infinite loop, also called an endless loop, is a sequence of instructions in a program which repeat indefinitely. . To terminate your program, just press Ctrl-C (or Ctrl-Break on some computers). This will cause a KeyboardInterrupt exception and let your program get out of the loop.

for i in range(100):

# do\_something()

 Pass

The for keyword opens the for loop; note – there's no condition after it; you don't have to think about conditions, as they're checked internally, without any intervention;

any variable after the for keyword is the control variable

note

**pass** -  keyword inside the loop body – it does nothing at all; it's an ****empty instruction.****

****break** – exits the loop immediately, and unconditionally ends the loop's operation; the program begins to execute the nearest instruction after the loop's body;**

****continue** – behaves as if the program has suddenly reached the end of the body; the next turn is started and the condition expression is tested immediately.**

**Both loops, while and for, have one interesting (and rarely used) feature.The loop's else branch,is always executed once, regardless of whether the loop has entered its body or not.**

**The range() function generates a sequence of numbers. It accepts integers and returns range objects. The syntax of range() looks as follows: range(start, stop, step), where:**

* **start is an optional parameter specifying the starting number of the sequence (0 by default)**
* **stop is an optional parameter specifying the end of the sequence generated (it is not included),**
* **and step is an optional parameter specifying the difference between the numbers in the sequence (1 by default.)**

**Ask chat**

**n = range(4)**

**for num in n:**

**print(num - 1)**

**else:**

**print(num)**

****Logic and Bit Operation****

****logical operators****. Logical operators take their arguments as a whole regardless of how many bits they contain. The operators are aware only of the value: zero (when all the bits are reset) means False; not zero (when at least one bit is set) means True. ****the logical operators do not penetrate into the bit level of its argument****.

**Conjunction  **and:**** depends on the fulfilment of these two conditions

**Disjunction **or :**** depends on at least one of these conditions

**not:**  It's a **unary operator performing a logical negation**.

De Morgan’s Law

**not (p and q) == (not p) or (not q)**

**not (p or q) == (not p) and (not q)**

****Bitwise operators** manipulate single bits of data.**

**Here are all of them:**

* **& (ampersand) ‒ bitwise conjunction; requires exactly two 1s to provide 1 as the result;**
* **| (bar) ‒ bitwise disjunction;  requires at least one 1 to provide 1 as the result;**
* **~ (tilde) ‒ bitwise negation; requires exactly one 1 to provide 1 as the result.**
* **^ (caret) ‒ bitwise exclusive or (xor).**

**the arguments of these operators **must be integers**; we must not use floats here.**

**Nb:**

****i = 15 i: 00000000000000000000000000001111****

****j = 22 j: 00000000000000000000000000010110****

****log = i and j ;** Both variables i and j are not zeros, so will be deemed to represent True. (True and True = True)**

****bit = i & j ;** The & operator will operate with each pair of corresponding bits separately, producing the values of the relevant bits of the result.**

1. ****Check the state of your bit,** nb:** a sequence of zeros and ones, whose task is to grab the value or to change the selected bits, is called a ****bit mask****

****x & 1 = x****

1. ****Reset your bit**** ‒ you assign a zero to the bit while all the other bits must remain unchanged  ****x &0 = 0****
2. **Set your bit** ‒ you assign a 1 to your bit, while all the remaining bits must remain unchanged; use the following disjunction property:
3. ****Negate your bit**‒ you replace a 1 with a 0 and a 0 with a 1. You can use an interesting property of the xor operator**

****x ^ 1 = ~x****

**x ^ 0 = x**

1. Python offers yet another operation relating to single bits: ****shifting****. This is applied only to ****integer**** values, and you mustn't use floats as arguments for it. The ****shift operators**** in Python are a pair of ****digraphs****: << and >>, clearly suggesting in which direction the shift will act. Eg: value << bits. ), ****shifting a value one bit to the left thus corresponds to multiplying it by two****; respectively, ****shifting one bit to the right is like dividing by two****

t an

10001

**LISTS**

**List** is a type of data in Python used to store multiple objects. It is an ordered and mutable collection of comma-separated items between square brackets (****may have different types****). the elements in a list are ****always numbered starting from zero**** (index).****Scalars**** variables that are able to store exactly one given value at a time.

***Accessing elements***

numbers = [10, 5, 7, 2, 1] , numbers[0] = = 10 , The value inside the brackets which selects one element of the list is called an index, while the operation of selecting an element from the list is known as indexing

Negative index: numbers[-1] == 1 , it starts with the last element

***List's elements may be removed***

del var [ index ] deletes the value at that index, whiles del var deletes the whole variable.

List.remove(‘value’) can also be used if specific element is known

**Functions vs. methods**

**A function** doesn't belong to any data ‒ it gets data, it may create new data and it (generally) produces a result.

***result = function(arg)***

**A method** does all these things, but is also able to change the state of a selected entity.The name of the method is preceded by the name of the data which owns the method. Next, you add a dot, followed by the method name, and a pair of parenthesis enclosing the arguments.

A method is owned by the data it works for, while a function is owned by the whole code.

***Adding elements to a list : list.append(value) ; list.insert(location, value)***

It takes its argument's value and puts it at the end of the list which owns the method.The **insert()** method is a bit smarter ‒ it can add a new element at any place in the list, not only at the end.

***for i in my\_list:***

***total += i***

easily **swap**

**my\_list = [10, 1, 8, 3, 5]**

**my\_list[0], my\_list[4] = my\_list[4], my\_list[0]**

**my\_list[1], my\_list[3] = my\_list[3], my\_list[1]**

**print(my\_list)**

***List methods***

***List.sort , list.reverse,list.index***

fruits = ["apple", "banana","cherry","apple"]

print(fruits.index("apple"))

***Operations on List***

The name of an ordinary variable is the name of its content; the name of a list is the name of a ***memory location( pointer)*** where the list is stored. In effect, if two lists (list\_1 and list\_2) identify the same location in the computer memory. Modifying one of them affects the other, and vice versa.

A **slice** is an element of Python syntax that allows you to ****make a brand new copy of a list, or parts of a list****.It actually copies the ***list's contents***, not the ***list's name***. ***ie :*** ***list\_2 = list\_1[:]***

***my\_list[start:end] : taking elements from the source list ‒ the elements of the indices from start to (end - 1*** ) .An element with an index equal to end is the first element which ****does not take part in the slicing****.

***my\_list[:end] == my\_list[0:end]*** If you omit the start in your slice, it is assumed that you want to get a slice beginning at the element with index 0.

***my\_list[start:] == my\_list[start:len(my\_list)]*** if you omit the end in your slice, it is assumed that you want the slice to end at the element with the index len(my\_list).

del my\_list[1:3]

**in and not in operators** : checks if an elements are in or not in a list

A two-dimensional array. It's also called, by analogy to algebraic terms, a matrix.

***List comprehension***allows you to create new lists from existing ones in a concise and elegant way. The syntax of a list comprehension looks as follows:

***[expression for element in list if conditional]***

which is the same as:

for element in list:

    if conditional:

        expression

Eg : ***board = [ [EMPTY for i in range(8)] for j in range(8) ]***

Objectives met

* Boolean values to compare different values and control the execution paths using the if and if-else instructions;
* the utilization of loops (while and for) and how to control their behavior using the break and continue instructions;
* the difference between logical and bitwise operations;
* the concept of lists and list processing, including the iteration provided by the for loop, and slicing;
* the idea of multi-dimensional arrays.

**Functions**

A function is a block of code that performs a specific task when the function is called (invoked).

There are at least four basic types of functions in Python:

* ***built-in functions*** which are an integral part of Python
* the ones that come from pre-installed ***modules***
* ***user-defined*** functions which are written by users for users
* the ***lambda*** functions
* **def *function\_name(parameter):***

***function\_body***

* ***function\_name(argument)***

**Why use functions:**

if a particular fragment of the code begins to appear in more than one place, consider the possibility of isolating it in the form of a function invoked from the points where the original code was placed before.

Each piece of code can be encoded separately, and tested separately. The process described here is often called***decomposition***

If a piece of code becomes so large that reading and understating it may cause a problem, consider dividing it into separate, smaller problems, and implement each of them in the form of a separate function.

*****You mustn't invoke a function which is not known at the moment of invocation.*****

**arguments exist outside functions**, and are carriers of values passed to corresponding parameters.

**You can pass arguments to a function using the following techniques:**

* **positional argument passing in which the order of arguments passed matters ,***positional arguments mustn't follow keyword arguments*****
* **keyword (named) argument passing in which the order of arguments passed doesn't matter,mustn't use a non-existent parameter name.**
* **a mix of positional and keyword argument passing**

You can ****pre-define**** a value for a given argument:

def name(first\_name, last\_name="Smith"):

print(first\_name, last\_name)

Example of key word argument in action:

def introduction(first\_name, last\_name):

print("Hello, my name is", first\_name, last\_name)

introduction(first\_name = "James", last\_name = "Bond")

introduction(last\_name = "Skywalker", first\_name = "Luke")

 If you try to pass more than one value to one argument, all you'll get is a runtime error.

def adding(a, b, c):

print(a, "+", b, "+", c, "=", a + b + c)

adding(3, a = 1, b = 2)

This is because a is already 3 , but we are also assigning it to 1

**Return**

**Functions and scopes**

The ****scope of a name**** (e.g., a variable name) is the part of a code where the name is properly recognizable. ***A variable existing outside a function has scope inside the function's body, excluding those which define a variable of the same name.*** A function receives the ****argument's value****, not the argument itself.

***global name***

***global name1, name2, …***

Using this keyword **global** inside a function with the name (or names separated with commas) of a variable (or variables), forces Python to refrain from creating a new variable inside the function ‒ the one accessible from outside will be used instead.

recursion is a ****technique where a function invokes itself****.

def fib(n):

if n < 1:

return None

if n < 3:

return 1

return fib(n - 1) + fib(n - 2)

def factorial\_function(n):

if n < 0:

return None

if n < 2:

return 1

return n \* factorial\_function(n - 1)

### Sequence types and mutability

A sequence type is a type of data in Python which is able to store more than one value (or less than one, as a sequence may be empty), and these values can be sequentially (hence the name) browsed, element by element. A sequence is data which can be scanned by the for loop

****mutability**** − is a property of any Python data that describes its readiness to be freely changed during program execution. There are two kinds of Python data: ****mutable**** and ****immutable****.

Nb: ***Situ >> mutable : )***

#### Tuples

Creating a tuple

***tuple\_1 = (1, 2, 4, 8)***

***tuple\_2 = 1., .5, .25, .125***

***empty\_tuple = ()***

***one\_element\_tuple\_1 = (1, )***

***one\_element\_tuple\_2 = 1.,***

***t = tuple(my\_list)***

each tuple element may be of a different type , accessing elements in a tuple is similar to lists indexing, negative indices and slicing.

* the len() function accepts tuples, and returns the number of elements contained inside;
* the + operator can join tuples together (we've shown you this already)
* the \* operator can multiply tuples, just like lists;
* the in and not in operators work in the same way as in lists.
* t1, t2, t3 = t2, t3, t1 they can  appear on the left side of the assignment operator
* a tuple's elements can be variables, not only literals.

*tup = 1, 2, 3, 2, 4, 5, 6, 2, 7, 2, 8, 9*

*duplicates = tup.****count(2)***

*print(duplicates) # outputs: 4*

**The dictionary** is a Python data structure.The dictionary uses ***keys and values*** It's ****not a sequence**** type (but can be easily adapted to sequence processing) and it is ****mutable****.  They don't preserve the order of their data

Creating a dictionary

***dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}***

***phone\_numbers = {'boss': 5551234567, 'Suzy': 22657854310}***

***empty\_dictionary = {}***

***colors = (("green", "#008000"), ("blue", "#0000FF"))***

***colors\_dictionary = dict(colors)***

****key-value**** pairs. Note:

* each key must be ****unique**** − it's not possible to have more than one key of the same value;
* a key may be ****any immutable type of object****: it can be a number (integer or float), or even a string, but not a list;
* a dictionary is not a list − a list contains a set of numbered values, while a ****dictionary holds pairs of values****;
* the len() function works for dictionaries, too − it returns the number of key-value elements in the dictionary;
* a dictionary is a ****one-way tool**** − if you have an English-French dictionary, you can look for French equivalents of English terms, but not vice versa.
* ***value = dictionary[‘key’] ;* **keys are case-sensitive****: 'Suzy' is something different from 'suzy'.
* **in**  and  **not in** operator; key in dictionary ; to verify

 keys(); method returns an iterable object consisting of all the keys gathered within the dictionary ***dictionary.keys()***

items().  returns tuples , where each tuple is a key-value pair. ***dictionary.items()***

***my\_dictionary.copy(), my\_dictionary.clear()***

**Modifying dictionary**

* dictionary['***key***'] = '***new value***' , this adds to the dictionary or updates a key if it already exist.
* dictionary***.update({ "duck" :  "canard" })***
* ***del  dictionary['key']***
* ***dictionary.popitem() ;  remove the last item in a dictionary,***

Nb: In the older versions of Python, i.e., before 3.6.7, the popitem() method removes a random item from a dictionary.

***for item in (d1, d2):***

***d3.update(item)***

***print(d3)***

Worth noting

my\_list = ['Mary', 'had', 'a', 'little', 'lamb']

def my\_list(my\_list):

del my\_list[3]

my\_list[3] = 'ram'

print(my\_list(my\_list))

a list named my\_list is created,

a function named my\_list is created,

the print function tries to invoke the my\_list function using the list my\_list as an argument. However, the list my\_list no longer exists because the function has the same name, and the function replaces the list,

the code will end in a runtime error because the function does not support item deletion.

PYTHON FOR DATA MANIPULATION

Numpy: Numerical Python, foundational pack‐ ages for numerical computing in Python. While NumPy by itself does not provide modeling or scientific functionality, having an understanding of NumPy arrays and array-oriented computing will help you use tools with array-oriented semantics, like pandas.

***Import numpy as np***

# creating numpy arrays ([ content ])

My\_arr = np.arange(10000) # arange is an array version Python range function

arr = np.arange(32).reshape((8, 4))

Data = np.random.randn(2,3) # generates some random data

Data1 = [1,2,3,4]

Data2 = [ [1,2,3] , [4,5,6] ]

Arr1 = np.array(data1)

Arr2 = np.array(data2)

np.zeros((3, 6)) # np.zeros(shape)

np.empty((2, 3, 2)) # , it may return uninitialized “garbage” values

zeros\_like,empty\_like,full\_like, eye, identity

# numpy data types

numerical dtypes are named the same way: a type name, like float or int, fol‐ lowed by a number indicating the number of bits per element

arr1 = np.array([1, 2, 3], dtype = ***np.float64*** )

***Data.dtype*** # Checking data type of an array

arr2 = arr1.astype(***np.int32***)

arr3 .astype(arr1.dtype)

# Basic Indexing and Slicing

Slices are views on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array.

one-dimensional objects such as Python lists, ndarrays can be sliced with the familiar syntax

[inclusive: excluding]

***1d***

arr = np.array([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])

arr[1:6] >> array([ 1, 2, 3, 4, 64])

***2d or nd (n dimensional)***

arr2d = np.array([ [1, 2, 3], [4, 5, 6], [7, 8, 9] ])

arr2d[:2] >>> array([ [1, 2, 3], [4, 5, 6] ])

arr2d[:2, 1:] >> array([ [2, 3], [5, 6] ])

arr2d[:2, 2] >>>: array([3, 6])

***Boolean indexing***

names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

data = np.random.randn(7, 4)

names == 'Bob' >>> array([ True, False, False, True, False, False, False], dtype=bool)

data[names == 'Bob'] # The boolean array must be of the same length as the array axis it’s indexing

data[names == 'Bob', 2:]

data[~(names == 'Bob')]

***Nb:*** Selecting data from an array by boolean indexing always creates a copy of the data, even if the returned array is unchanged.

The Python keywords and and or do not work with boolean arrays. Use & (and) and | (or) instead.

***Fancy indexing***

a term adopted by NumPy to describe indexing using integer arrays

Data[ [r1,r2,r6] , [c1,c3,c4] ]

arr = np.empty((8, 4))

for i in range(8): arr[i] = I

arr[ [4, 3, 0, 6] ]

>>>

array([ [ 4., 4., 4., 4.], [ 3., 3., 3., 3.], [ 0., 0., 0., 0.], [ 6., 6., 6., 6.] ])

***Universal function :*** ufunc performs element-wise operations

on data in ndarrays

np.sqrt(arr)

np.exp(arr), np.maximum(x, y)

result = np.where(cond, xarr, yarr)

np.where(arr > 0, 2, arr) # set only positive values to 2

***Mathematical and Statistical Methods***

arr.mean(), np.mean(arr)

arr.sum()

arr.mean(axis=1)

arr.sum(axis=0)

arr.mean(1) means “compute mean across the columns” where arr.sum(0) means “compute sum down the rows.”

bools.any(),bools.all()

any tests whether one or more values in an array is True, while all checks if every value is True

arr.sort()

np.unique(names) #np.unique, which returns the sorted unique values in an array:

***File Input and Output with Arrays***

np.save and np.load are the two workhorse functions for efficiently saving and load‐ ing array data on disk.

arr = np.arange(10)

np.save('some\_array', arr)

np.load('some\_array.npy')

For saving multiple values

np.savez('array\_archive.npz', a=arr, b=arr)

arch = np.load('array\_archive.npz')

arch['b']

Data.shape