| **MAIN TOPIC** | **SUB TOPIC** | **DETAILS**  **/**  **(LINKS FOR FURTHER STUDY)**  **/**  **(FEEDBACK)**  **/**  **(SAMPLE PROGRAMS)** | **(CLASSROOM EXERCISES)**  **/**  **(ASSIGNMENTS)** | **TRACKING INFORMATION** |
| --- | --- | --- | --- | --- |
| **OVERALL CONTEXT** | GOALS AS A TRAINER | * Teach the basics of Python in an informal and interactive way * Ensure that the course content is covered within the given time frame * Get you motivated to start programming * Get you curious about the language. Python is fun to program with !. * Ensure that you are generally attentive during the course * Get positive feedback at the end of the course | **N/A** | DAY 1  (<= 15 mins) |
|  | PROPOSED TEACHING STYLE | * Interactive / personalized style with lots of visuals   i.e. why I use a WORD document instead of a PPT   * Avoid the use of lengthy paragraphs * Avoid spending too much time talking on a specific topic * Switching contexts too often – Strict NO NO * Get all of you to write lots of programs * Will try to use images, because our brain understands visuals more easily than text | **N/A** | DAY 1  (<= 15 mins) |
|  | MY EXPECTATIONS FROM THE STUDENTS/YOU | * Be aware of the course content (*Have all of you gone through the course details ?)* * Do the class room exercises * Make notes * Don’t just read what I say. Digest it slowly. Stop me if I am going too fast | **N/A** | DAY 1  (<= 15 mins) |
|  | WHAT ARE YOU EXPECTING ? | <Update after feedback from the students> | **N/A** | DAY 1  (<=15 mins) |
|  | ABOUT THE TRAINER I.E MYSELF |  | **N/A** | DAY 1  (<= 15 mins) |
|  | NEXT STEPS | We will revisit this topic at the end of the course | **N/A** | DAY 1  (<= 15 mins) |
|  | ANY OTHER POINTS ? | * <Update after feedback from the students> | **N/A** | DAY 1  (<= 15 mins) |
| **PYTHON ECOSYSTEM** | PYTHON INSTALLATION related information | * For the course, we will be using Python 3.7.x * Download link <https://www.python.org/downloads/> * Find out if there are any * Mac users * Linux users |  | DAY 1  (<= 30 mins) |
|  | Environment variables that need to be set | * Windows   + Path   + PYTHON\_HOME |  | DAY 1  (<= 10 mins) |
|  | Installing third party editors | * Notepad++ link –   <https://notepad-plus-plus.org/download/v7.5.9.html>   * WingWare Link   <https://wingware.com/downloads/wingide-101>   * PyCharm   <https://www.jetbrains.com/pycharm/download/#section=windows> |  | DAY 1  (<= 30 mins) |
|  | Brief history of Python | * Source (there are actually many) * <https://www.python-course.eu/python3_history_and_philosophy.php> * <https://medium.com/@johnwolfe820/a-brief-history-of-python-ca2fa1f2e99e> |  | DAY 1  (<= 15 mins) |
|  | Python’s rich standard library | * The standard library is the jewel in Python’s crown, supplying reusable modules that help you with everything from, for example, working with data, through manipulating ZIP archives, to sending emails, to working with HTML. * The standard library even includes a web server, as well as the popular SQLite database technology * During the course, we will be using the standard library extensively |  | DAY 1  (<= 30 mins) |
|  | Third party modules | Python has a rich collection of third party modules that developers can use in their applications.  A few well known modules :-   * Web Development * Django * Flask * Data Analysis * Numpy * Pandas * Machine Learning * PyTorch * HTTP Library * Requests * Automated Unit testing * PyTest * Interactive IDEs * Jupyter * Database Toolkits * SQLAlchemy * Web scraping * Scrapy |  | DAY 1  (<= 30 mins) |
|  | The Python Interpreter | * When Python is installed, the interpreter gets installed * There is no “EXE” in Python. * When you run a program, the interpreter compiles the code and executes the compiled code * Python *<<program/module name>>* |  | DAY 1  (<= 10 mins) |
|  | Python’s interactive shell | * Executing just python opens up the interactive shell * Run “python” from the command line * >>> appears |  | DAY 1  (<= 10 mins) |
|  | Different editors/ IDEs | * IDLE – This is Python’s default editor * WingWare (Will not be covered in the course) * Notepad/Notepad++ * PyCharm (Will not be covered in the course) * Jupyter Notebooks (will not be covered in the course) |  | **DAY 1**  **(<= 10 mins)** |
|  | What are people using Py for ? | Earlier we discussed the standard library and third party modules . This gives a good indication about what Python is being used for   * Web development * Web services * Web scraping * Batch processing * Data sciences * Data processing * Machine Learning * Big data   What’s missing from the list ?   * Mobile application development ! |  | **DAY 1**  **(<= 10 mins)** |
|  | ANY OTHER POINTS ? | * Debuggers - pdb - (Will not be covered in this course)   + Basic info in *<< pibm-ips-images-other-stuff.docx>>* |  | **DAY 1**  **(<= 10 mins)** |
|  | Learning to use the interactive shell | * Used to run snippets of Python code * Generally used for executing a single statement at a time * Helpful when starting to learn Python * More info in <<py\_interactive\_shell\_the\_basics.py>> |  | **DAY 2**  **(<=1 hr)** |
|  | Learning to use IDLE – Python’s default editor | * Launching IDLE * <Python\_Home>\Lib\idlelib\idle.bat   *(Python\_home is the environment variable)* |  | **DAY 2**  **(<=30 mins)** |
|  | WingWare IDE | * Time permitting, we might look at this IDE later   *(There is no one size fits “ALL”. Developers use a wide variety of IDEs for developing Python apps.)*  Always choose one that you are comfortable with. |  | **DAY 2** |
| **PYTHON BASIC PROGRAMMING** | What do you think a Python program contains ? |  |  | **DAY 2**  **(<= 30 mins)** |
|  | In general, what do you think programs should do ? | * Handle data * Process input to generate output * Over the years, languages have evolved to manage complex data * A programming language must have data structures to be able to efficiently process information |  | **DAY 2**  **(<= 30 mins)** |
|  | Some basic code samples | *Python <program name>*   * Really\_simple \_example.py * Another\_simple\_example.py * <Create a few more at run time> |  | **DAY 2**  **(<= 30 mins)** |
|  | Declaring variables | * declaring\_variables\_example\_001.py * declaring\_variables\_example\_002.py * declaring\_variables\_example\_003.py * declaring\_variables\_example\_004.py * declaring\_variables\_example\_005.py * declaring\_variables\_example\_006.py   A few points to be noted   * + The data type is not explicitly stated   + A variable can be assigned different types of values   (This feature will be explained in detail when we talk about everything is an object) | **PY-CL-DECLARING-VARIABLES** | **DAY 2**  **(<= 1 hr)** |
|  | More on declaring String variables | * In most programs, strings are very widely used * Python provides extensive features for dealing with Strings * Link for further reading - <https://docs.python.org/3/library/stdtypes.html#string-methods> * Sample program - decl\_string\_var\_example\_001.py | **PY-CL-DECLARING-STRING-VARIABLES** | **DAY 2**  **(<= 30 mins)** |
|  | Can you think of any other variables that you would like to declare in your programs ? | * Large numbers with decimals * Arrays * Multi dimensional arrays * Data structures to handle non structured information that today’s applications generate |  | **DAY 2** |
|  | CLASS ROOM EXERCISES |  |  | **DAY 2** |
|  | ANY OTHER POINTS ? | <Update after feedback from students> |  | **DAY 2** |
|  | **EVERYTHING IS AN OBJECT** | * This is something that you will read about all the time * We will, over the next few weeks, whenever required, re-visit this topic * Each language has its own way of interpreting what an object actually is * More details in *<< pibm-ips-images-other-stuff.docx >>* * Sample code * py\_everything\_is\_an\_obj\_001.py * py\_everything\_is\_an\_obj\_002.py * py\_everything\_is\_an\_obj\_003.py * py\_everything\_is\_an\_obj\_004.py   This might not be fully understood at this stage.   * More sample programs for mutable types * everything\_is\_an\_obj\_example\_001.py * everything\_is\_an\_obj\_example\_002.py   DON”T WORRY, we will continue to discuss this important topic in later sessions. | * PY-CL-VARIABLES-OBJECTS   + Declare 5 integer variables, set the value to 1   Use id and type functions for all 5, what do you observe ?   * + Use the above program   Add the following code -  Set each of the 5 variables to “pibm”  Use id and type functions for all 5, Now, what do you observe ? |  |
| **THE NEXT LEVEL OF PYTHON PROGRAMS** | In your opinion what can we cover next ? | <update after feedback from students> |  |  |
|  | INDENTATION | * Every language has its own syntax * There is a specific way to identify statements, a group of statements and blocks of code * Examples are “;”, “{“, “}”, etc.   Python’s approach is very different and is referred to as “INDENTATION”. Please refer to *<< pibm-ips-images-other-stuff.docx>>*   * Indentation is something that is both unique and peculiar to Python * How to statements end ? * What is a block of code ?   Some bad indentation examples   * indentation\_bad\_example\_001.py * indentation\_bad\_example\_002.py * indentation\_bad\_example\_003.py |  |  |
|  | If | * We will learn to create decisions in a Python program using the various forms of “if” * Learn to conditionally execute code in a program * Sample code - <<If\_example\_001.py>> |  |  |
|  | If else | * Sample code - <<if\_else\_example\_001.py>> * if\_else\_ladder\_example\_001.py   (Avoid writing such code !) |  |  |
|  |  |  |  |  |
|  | If elif | * Sample code - << if\_elif\_example\_001.py >> * Sample code - << if\_elif\_nesting\_example\_001.py >> |  |  |
|  | WHILE loops | * Sample code - << while\_loop\_example\_001.py >> * Sample code - << while\_loop\_with\_else\_example\_001.py >> |  |  |
|  | FOR Loops | * This is very widely used in Python * The for loop in Python is used to iterate over a sequence (Examples - [list](https://www.programiz.com/python-programming/list), [tuple](https://www.programiz.com/python-programming/tuple), [string](https://www.programiz.com/python-programming/string)) or other **iterable** objects. * Iterating over a sequence is called traversal.   Let’s take a step back and discuss the concept of iteration   * Sample code - << for\_loop\_example\_001.py >> * Sample code - << for\_loop\_example\_002.py >> * Sample code - << for\_loop\_example\_003.py >> * <<for\_loop\_example\_004.py>> |  |  |
|  | CLASSROOM EXERCISES |  |  |  |
|  | ANY OTHER POINTS TO DISCUSS |  |  |  |
| **DATA STRUCTURES** | Scratching the surface |  |  |  |
|  | Tuples |  |  |  |
|  | Lists |  |  |  |
|  | Dictionaries |  |  |  |
|  | Sets |  |  |  |
|  | CLASSROOM EXERCISES |  |  |  |
|  | Any other points to discuss ? |  |  |  |
| * **D.R.Y.** * **MODULARITY** * **WRITING BETTER CODE** | Using DocString for documentation |  |  |  |
|  | Functions in Python | What is DRY ?  Let’s look at some really bad programs |  |  |
|  | Exception handling |  |  |  |
|  | Coding standards |  |  |  |
|  | Static code Analysis |  |  |  |
|  | Modules and Packages |  |  |  |
|  | A quick visit back to the house of the “Standard Library” |  |  |  |
| **DATA STRUCTURES** | Diving below the surface |  |  |  |
| **OOPS IN PYTHON** |  |  |  |  |
|  |  |  |  |  |
| **CONCLUDING THE COURSE** | What went well ? |  |  |  |
|  | So – so ..ummm |  |  |  |
|  | Really bad |  |  |  |
|  | Next steps |  |  |  |

1. What’s next for you

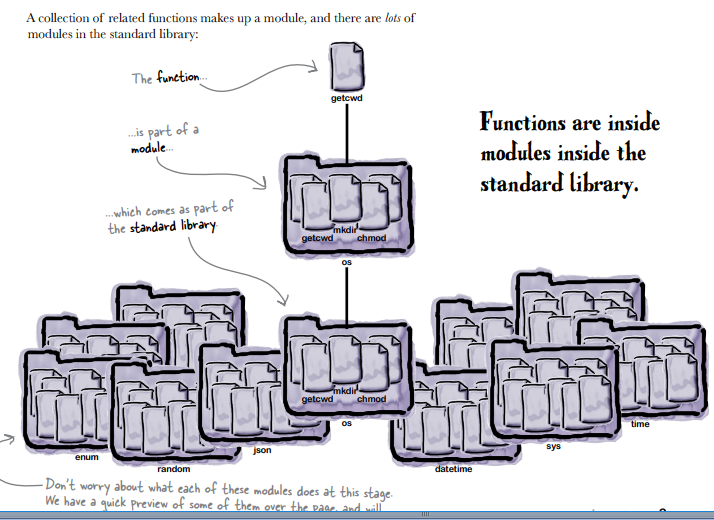
* The first window, the Python Shell, is a REPL environment used to run snippets of Python code, typically a single statement at a time. The more you work with Python, the more you’ll come to love the Python Shell, and you’ll be using it a lot as you progress through this book. For now, though, we are more interested in the second window.

|  |
| --- |
| >>> dir(random)  >>> help(random.randint) |

|  |
| --- |
| • Debuggers. A debugger allows a programmer to more easily trace a program’s execution in order to locate and correct errors in the program’s implementation. With a debugger, a developer can simultaneously run a program and see which line in the source code is responsible for the program’s current actions. The programmer can watch the values of variables and other program elements to see if their values change as expected. Debuggers are valuable for locating errors (also called bugs) and repairing programs that contain errors.  NOT BEING COVERED IN THE COURSE - PDB |

>> Do you want to introduce the Standard Library – YES

>> Include a diagram similar to this



>> Think of modules *(Python refers to programs as modules)* as a collection of related functions.

>> The standard library is the jewel in Python’s crown, supplying reusable modules that help you with everything from, for example, working with data, through manipulating ZIP archives, to sending emails, to working with HTML. The standard library even includes a web server, as well as the popular SQLite database technology. {{ This is copied text re-word }}

>> .The Python documentation has all the answers on the standard library. Here’s the kicking-off point:

https://docs. python.org/3/library/index.html

>> The standard library isn’t the only place you’ll find excellent importable modules to use with your code. The Python community also supports a thriving collection of third-party modules, some of which we’ll explore later in this book. If you want a preview, check out the community-run repository: <http://pypi.python.org>. {{ This is copied text re-word }}

>> THEME : Data Structures Come Built-in { I will refer to this repeatedly } The focus is on covering the course content, but given the power of data structures, we will explore more advanced concepts together

**>> PAUSE PAUSE PAUSE**

>> In our sample programs and assignments, we will at times be using the standard library

>> Please feel free to explore the standard library [[ Assignment for the group ]]

>> Python variables are dynamically assigned Before getting to the next line of code, perhaps a few words are needed about variables, especially if you are one of those programmers who might be used to pre declaring variables with type information before using them (as is the case in statically typed programming languages). In Python, variables pop into existence the first time you use them, and their type does not need to be predeclared. Python variables take their type information from the type of the object they’re assigned.

{{ How and When DO I introduce this concept }}

>> We will write programs and when required using the Interactive Shell

>> Let’s dive in

>> Let’s execute the simplest program possible using IDLE - **Really-Simple.py**

>> We could have done this directly from the Shell itself using the print function

>> The shell helps you run snippets of code - The interactive interpreter is most useful for experimenting with small snippets of Python code

>> As you get more and more experienced, as a developer you will execute programs

>> One can also run this from the command line

python .\sample-programs\Really\_Simple\_example.py

>> More on the Interactive Shell

>>> help() {{ The prompt at the interactive shell changes to >>>, remember this }}

>> Let’s pause and summarize what has been covered so far

* Interpreter
* Interactive Shell
* IDLE
* And a very simple program

>> From you, I would like to know what are the basic features that Python needs to provide

* Variables to hold data
* Statements
* If statements
* Looping

|  |
| --- |
| What is a compiler?  2. What is an interpreter?  3. How is a compiler similar to an interpreter? How are they different?  4. How is compiled or interpreted code different from source code?  5. What tool does a programmer use to produce Python source code?  6. What is necessary to execute a Python program?  7. List several advantages developing software in a higher-level language has over developing software in machine language.  8. How can an IDE improve a programmer’s productivity?  9. What is the “official” Python IDE?  10. What is a statement in a Python program? |

**>> Let’s start coding**

* **Indentation rules [ specific to Python] – Walkthrough at least 4 programs**
* **Constructs**

>> Program with some variables - var-first-try.py

>> Program with different kinds of variables - var-second-try.py

>> Messing around with your program

>> Typically in training programs, working code is used. We will look at plenty of programs that don’t work and fix them.

>> Any simple programs – invite suggestions from the group

>> A program that does nothing

>>

|  |
| --- |
| Identifiers While mathematicians are content with giving their variables one-letter names like x, programmers should use longer, more descriptive variable names.  Names such as sum, height, and sub\_total are much better than the equally permissible s, h, and st. A variable’s name should be related to its purpose within the program.  Good variable names make programs more readable by humans. Since programs often contain many variables, well-chosen variable names can render an otherwise obscure collection of symbols more understandable. Python has strict rules for variable names.  A variable name is one example of an identifier.  An identifier is a word used to name things.  One of the things an identifier can name is a variable.  We will see in later chapters that identifiers name other things such as functions, classes, and methods.  Identifiers have the following form:  • An identifiers must contain at least one character.  • The first character of an identifiers must be an alphabetic letter (upper or lower case) or the underscore ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz\_  • The remaining characters (if any) may be alphabetic characters (upper or lower case), the underscore, or a digit ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz\_0123456789  • No other characters (including spaces) are permitted in identifiers.  • A reserved word cannot be used as an identifier |

>> So far, we have created simple programs which handle data in a really simple way

>> integers, floating point numbers, strings, text

|  |
| --- |
| Exercises  Which is better, too many comments or too few comments  What is the purpose of comments?  Why is human readability such an important consideration? |

|  |
| --- |
| How many spaces should you indent? Python requires at least one, some programmers consistently use two, four is the most popular number, but some prefer a more dramatic display and use eight. A four space indentation for a block is the recommended Python style. This text uses the recommended four spaces to set off each enclosed block. In most programming editors you can set the Tab key to insert spaces automatically so you need not count the spaces as you type. Whichever indent distance you choose, you must use this same distance consistently throughout a Python program.  Why is indentation that mixes tabs and spaces a problem and thus forbidden in Python 3? Consider creating a Python source file in one editor and then viewing it in a different editor with tab stops set differently. Lines that appear perfectly indented in the original editor would be misaligned in the new editor. Instead, code indented with four spaces within one editor would appear exactly the same in any other editor. Python 3 does allow the use of tabs for indentation—you just cannot mix them with spaces within the same source file. |

|  |
| --- |
| #simple\_if.py  # Get two integers from the user  dividend = int(input('Please enter the number to divide: '))  divisor = int(input('Please enter dividend: '))  # If possible, divide them and report the result  if divisor != 0:  quotient = dividend/divisor  print(dividend, '/', divisor, "=", quotient)  print('Program finished') |
| #simple\_if.py  # Will not work  # Get two integers from the user  dividend = int(input('Please enter the number to divide: '))  divisor = int(input('Please enter dividend: '))  # If possible, divide them and report the result  if divisor != 0:  quotient = dividend/divisor  print(dividend, '/', divisor, "=", quotient)  print('Program finished') |

>> deciding when to run blocks of code

|  |
| --- |
| Simple if |
| If else |
| If elif else  // Python provides a multi-way conditional construct called if/elif/else that permits a more manageable textual structure for programs that must check many conditions |

|  |
| --- |
| 18. Write a Python program that requests five integer values from the user. It then prints the maximum and minimum values entered. If the user enters the values 3, 2, 5, 0, and 1, the program would indicate that 5 is the maximum and 0 is the minimum. Your program should handle ties properly; for example, if the user enters 2, 4 2, 3 and 3, the program should report 2 as the minimum and 4 as maximum. |
| 19. Write a Python program that requests five integer values from the user. It then prints one of two things: if any of the values entered are duplicates, it prints "DUPLICATES"; otherwise, it prints "ALL UNIQUE". |

>> Iteration

|  |
| --- |
| // What do you think of this code  print(1)  print(2)  print(3)  print(4)  print(5)  print(6)  print(7)  print(8)  print(9)  print(10) |

ITERATION TO THE RESCUE

>> Using iteration can greatly simplify programs

>> Using iteration helps you to loop/traverse through data sets

|  |
| --- |
| While |
| For  // Python’s for loop is amazing … iterating through arbitrary data sets  // See below |
| RANGE function  **range**(*stop*)  **range**(*start*, *stop*[, *step*])  <https://docs.python.org/3/library/stdtypes.html#typesseq-range> |
| WHILE/ELSE AND FOR/ELSE |
| BREAK/CONTINUE |

|  |
| --- |
| for ch in "Hi!":  print(ch) |
| for i in [1, 2, 3]:  print(i) |
| |  | | --- | | Python’s for statement is built to work with an iterable object | |
| Using the in / not in operators – This is quite useful in Python |

|  |
| --- |
| Exercises |

>> PAUSE PAUSE PAUSE PAUSE Exercises

>> We have been running fairly straight forward programs

>> Let’s use the wonderful standard library as well

|  |
| --- |
| Interactive Shell  >>> dir(random)  >>> help(random.randint) |

|  |
| --- |
| **FOR LATER FOR LATER FOR LATER** |
| “Everything Is an Object” Any object can be dynamically assigned to any variable in Python. Which begs the question: what’s an object in Python? The answer: everything is an object. All data values in Python are objects, even though—on the face of things—“Don’t panic!” is a string and 42 is a number. To Python programmers, “Don’t panic!” is a string object and 42 is a number object. Like in other programming languages, objects can have state (attributes or values) and behavior (methods)  All this talk of “objects” can mean only one thing: Python is objectoriented, right?  Sort of. You can certainly program Python in an object-oriented way using classes, objects, instances, and so on (more on all of this later in this book), but you don’t have to. Recall the programs from the last chapter...none of them needed classes. Those programs just contained code, and they worked fine. Unlike some other programming languages (most notably, Java), you do not need to start with a class when first creating code in Python: you just write the code you need. Now, having said all that (and just to keep you on your toes), everything in Python behaves as if it is an object derived from some class. In this way, you can think of Python as being more objectbased as opposed to purely object-oriented, which means that object-oriented programming is optional in Python. |

>> Data Structures

>> We are going to spend a lot of time on this

>> Any serious program needs to handle data. A good programming language must provide data structures that will allow you to map diverse data sets to efficient data structures.

>> Data sciences

|  |
| --- |
| Python comes with four built-in data structures that you can use to hold any collection of objects, and they are list, tuple, dictionary, and set. [ LTDS] |
| Note that by “built-in” we mean that lists, tuples, dictionaries, and sets are always available to your code and they do not need to be imported prior to use: each of these data structures is part of the language. |

|  |
| --- |
| **List:** an ordered mutable collection of objects A list in Python is very similar to the notion of an array in other programming languages, in that you can think of a list as being an indexed collection of related objects, with each slot in the list numbered from zero upward. Unlike arrays in a lot of other programming languages, though, lists are dynamic in Python, in that they can grow (and shrink) on demand. There is no need to predeclare the size of a list prior to using it to store any objects. Lists are also heterogeneous, in that you do not need to predeclare the type of the object you’re storing—you can mix’n’match objects of different types in the one list if you like. Lists are mutable, in that you can change a list at any time by adding, removing, or changing objects. |
| Tuple: an ordered immutable collection of objects A tuple is an immutable list. This means that once you assign objects to a tuple, the tuple cannot be changed under any circumstance. It is often useful to think of a tuple as a constant list. |
| Dictionary: an unordered set of key/value pairs Depending on your programming background, you may already know what a dictionary is, but you may know it by another name, such as associative array, map, symbol table, or hash. Like those other data structures in those other languages, Python’s dictionary allows you to store a collection of key/value pairs. Each unique key has a value associated with it in the dictionary, and dictionaries can have any number of pairs. The values associated with a key can be any object. Dictionaries are unordered and mutable. It can be useful to think of Python’s dictionary as a two-columned, multirow data structure. Like lists, dictionaries can grow (and shrink) on demand. |
| Set: an unordered set of unique objects In Python, a set is a handy data structure for remembering a collection of related objects while ensuring none of the objects are duplicated. The fact that sets let you perform unions, intersections, and differences is an added bonus (especially if you are a math type who loves set theory). Sets, like lists and dictionaries, can grow (and shrink) as needed. Like dictionaries, sets are unordered, so you cannot make assumptions about the order of the objects in your set. As with tuples and dictionaries, you’ll get to see sets in action in the next chapter. |

>> Please include diagrams

>> Once again remember everything is an object

>> Iterations in the context of data structures

>> Lets talk about real world data

>> How do we map this to data structures

>> 80:20 rule

>> Do these 4 structures cover all the cases – NO

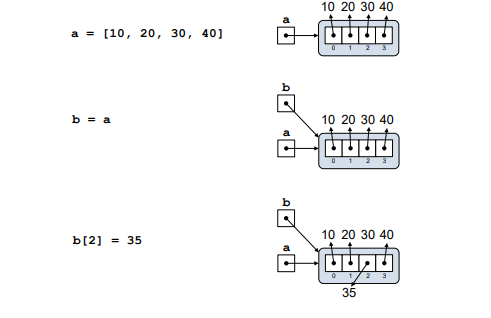
|  |
| --- |
| The four built-in data structures are useful, but they don’t cover every possible data need. However, they do cover a lot of them. It’s the usual story with technologies designed to be generally useful: about 80% of what you need to do is covered, while the other, highly specific, 20% requires you to do more work. |

>> Examples – Some of these can be really strange

|  |
| --- |
| TUPLES – () |
| Tuples are pretty easy to make. You give your tuple a name, then after that the list of values it will carry. For example, the months of the year: |
| months = ('January','February','March','April','May','June',\  'July','August','September','October','November',' December')  Index notation 0 to 11 |
| Cats = ( 'Tom', 'Snappy', 'Kitty', 'Jessie', 'Chester' )  Index notation = 0 to 4 |
| Relay\_Team = ( (‘jack’, ‘Jim’, “john’, ‘jake’), ('Tom', 'Snappy', 'Kitty', 'Jessie')) |
| t = (1, 2, "a", “delta”, true) |
| Considered to be a sequence type |

|  |
| --- |
| Exercises |

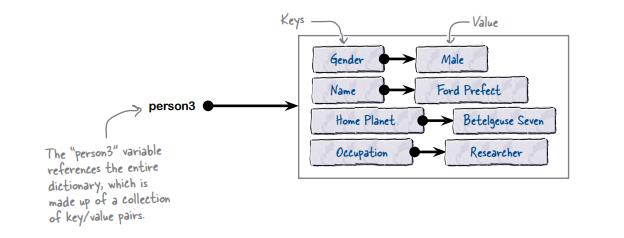
|  |
| --- |
| LISTS IN PYTHON |
| lst = [2, -3, 0, 4, -1] |
| a = [] //empty list |
| col = [23, [9.3, 11.2, 99.0], [23], [], 4, [0, 0]]  // heterogenous elements in the list |
| Negative indexes |
| a = [2, 4, 6, 8]  a = a +[1,2,3] |
| Membership in a list |
| Lists and tuples are convenient for storing collections of data, but they have some limitations. For one, we locate an element within a list or tuple based on its position (index). While this approach is fine for many applications, in other situations this access-by-index approach is awkward or inefficient. |
| Great for collections of similar objects |
| Considered to be a sequence type |



**DICTIONARIES**

|  |
| --- |
| A Python dictionary is an associative container which permits access based on a key, rather than an index. |
| In a dictionary every key has an associated value. |
| Ideal for key value pairs |
| A dictionary key may be of any immutable type. This means all of the following can serve as keys within a dictionary: integers, floating-point numbers, strings, Booleans, and tuples. |
| Since lists are mutable objects, a list may not be a key. A dictionary is a mutable object, so a dictionary cannot use itself or another dictionary object as a key |
| d = {'Fred': 44, 'Ella': 39, 'Owen': 40, 'Zoe': 41} |
| The order of key:value pairs in a dictionary are independent of the order of their insertion into the dictionary  Insertion order is not maintained |
|  |
| You should use a dictionary when you need fast and convenient access to an element of a collection based on a search key rather than an index |
|  |
| Sorted function |
| A complex structure that comes up a lot is a dictionary of dictionaries. This structure can be used to create a mutable table. |

Good image to explain dictionaries



**SETS**

|  |
| --- |
| Python provides a data structure that represents a mathematical set |
| . As with mathematical sets, we use curly braces ({}) in Python code to enclose the elements of a literal set. |
| Python distinguishes between set literals and dictionary literals by the fact that all the items in a dictionary are colon-connected (:) key-value pairs, while the elements in a set are simply values. |
| Unlike Python lists, sets are unordered and may contain no duplicate elements. |
|  |

**HANDLING EXCEPTIONS IN CODE**

|  |
| --- |
| In our programming experience so far we have encountered several kinds of run-time exceptions, such as division by zero, accessing a list with an out-of-range index, and attempting to convert a non-number to an integer. |
| We have seen these and other run-time exceptions immediately terminate a running program |
| Python provides a standard mechanism called exception handling that allows programmers to deal with these kinds of run-time exceptions and many more |
| Rather than always terminating the program’s execution, an executing program can detect the problem when it arises and possibly execute code to correct the issue or mitigate it in some way. |
|  |
| Python’s exception handling infrastructure allows programmers to cleanly separate the code that implements the focused algorithm from the code that deals with exceptional situations that the algorithm may face. |
| This approach is more modular and encourages the development of code that is cleaner and easier to maintain and debug. |
| Things go wrong no matter how good your code is |

|  |
| --- |
| Examples of code that generate run time exceptions |
| Dividing by zero  Trying to add a String to a number |
|  |
|  |
|  |

Functions and Modules

|  |
| --- |
| DRY …DRY …DRY…. |
| Building modular, easy to maintain system |
| Enabling code reuse |
| Breaking large chunks of code into smaller modules |
|  |
| Functions are designed to return a single value |
| Python lets you send any *object* as a argument, and pass back any *object* as a return value. The interpreter doesn’t care or check what type these objects are (only that they are  provided). |
|  |
| The only way to return multiple values is to package the multiple values in a single data structure, then return that. |
| A Python module is simply a file that contains Python code. |
| from *m o d u le* import *fu n c ti o n l is t* |
| from math import \*  makes all the code in the math module available to the program.  // Strongly discouraged , name collision |
| import math - uses qualified names for the functions the module provides |
| import math as m  In this case, the caller would invoke the modules functions in the following way:  y = m.sqrt(x)  print(m.log10(100)) |

|  |
| --- |
| CONTEXT MANAGERS |
| FUNCTION DECORATORS |
|  |