

Introduction to Python

Last Week

1. Get Python Running
2. The Command-Line, Directories & Files
3. What is Python
4. Variables
5. Input/Output
6. Data Types - Numbers: int, float, complex
7. Data Types - Text: str
8. Data Types - Sequences: lists, tuples, range
9. Control Flow & Compound Statements: if, for while
10. Simple First Programs

Objectives of this Lesson

1. Set up an Integrated Development Environment (IDE)
2. Data Type - Mapping: dictionaries
3. Functions
4. Variable Scope
5. File I/O
6. Programming with Functions

Setting up Atom IDE

Download @ <https://atom.io/>

Installation Guide @

[http://flight-manual.atom.io/getting-started/sections/installing-atom/#platform-ma](http://flight-manual.atom.io/getting-started/sections/installing-atom/#platform-mac)

[C](http://flight-manual.atom.io/getting-started/sections/installing-atom/#platform-windows)

<http://flight-manual.atom.io/getting-started/sections/installing-atom/#platform-windows>

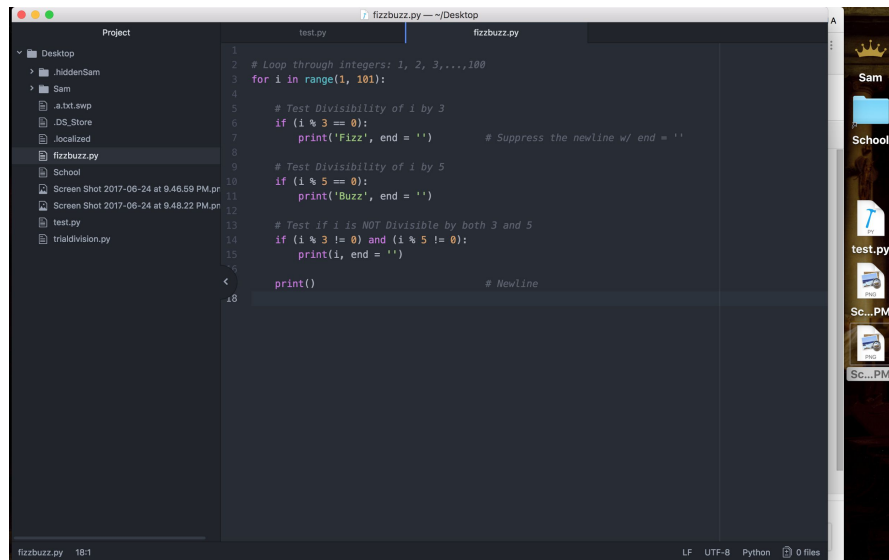
<http://flight-manual.atom.io/getting-started/sections/installing-atom/#platform-linux>

List of Other Python IDEs:

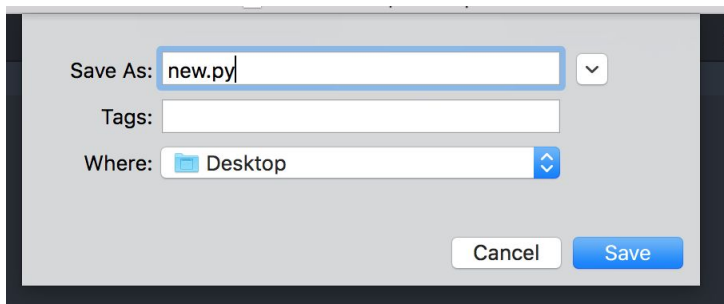
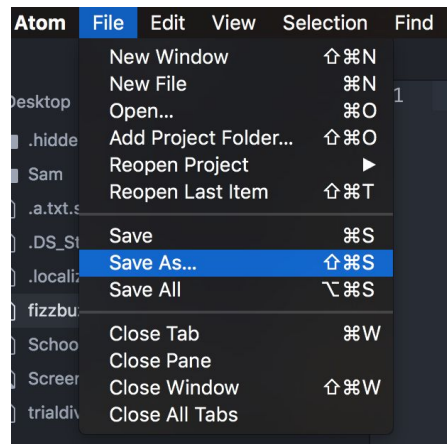
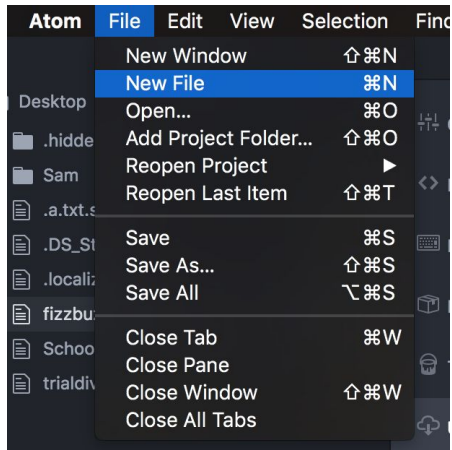
https://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments#Python
[on](#)

Important Features:

- >Syntax highlighting
- >Indentation
- >Autocomplete
- >Brace matching



Creating a New .py File



Running a .py Script

- 1) Navigate to the directory containing the .py script you wish to run
- 2) Run the script with this command:

`python[3] yourscrip.py`

```
[slr:~ sammishra$ pwd
/Users/sammishra
[slr:~ sammishra$ cd Desktop/
[slr:Desktop sammishra$ pwd
/Users/sammishra/Desktop
[slr:Desktop sammishra$ ls -l
total 2496
drwxr-xr-x@ 10 sammishra  staff      340 Jun 23 13:06 Sam
-rw-r--r--@  1 sammishra  staff 1268892 Jun 23 13:06 School
-rw-r--r--@  1 sammishra  staff   385 Jun 24 21:46 fizzbuzz.py
-rw-r--r--@  1 sammishra  staff   235 Jun 24 10:43 trialdivision.py
[slr:Desktop sammishra$ python3 trialdivision.py
```

Objectives of this Lesson

1. Set up an Integrated Development Environment (IDE)
2. Data Type - Mapping: dictionaries
3. Functions
4. Variable Scope
5. File I/O
6. Programming with Functions

Data Types - Mapping: Dictionary

A mapping object **maps hashable values to arbitrary objects** (keys to data).

Mappings are **mutable** objects.

There is only one standard mapping type, the **dictionary**.

Dictionaries can be created by the dict constructor placing a comma-separated list of **key: value** pairs within braces, for example:

```
{ 'jack': 4098, 'sjoerd': 4127 } or { 4098: 'jack', 4127: 'sjoerd' }
```

Dictionaries compare **equal**, if and only if, they have the same **key: value** pairs.

Order comparisons ('<', '<=', '>=', '>') raise TypeError.

Data Types - Mapping: Dictionary Constructor

```
class dict(**kwarg)
```

```
class dict(mapping, **kwarg)
```

```
class dict(iterable, **kwarg)
```

If no positional argument is given, an empty dictionary is created.

If a positional argument is given and it is a mapping object, a dictionary is created with the same key-value pairs as the mapping object.

Otherwise, the positional argument must be an iterable object.

If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

The following examples all return a dictionary equal to `{"one": 1, "two": 2, "three": 3}`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

Data Types - Mapping: Dictionary Functions

`len(d)`

Return the number of items in the dictionary *d*.

`d[key]`

Return the item of *d* with key *key*. Raises a `KeyError` if *key* is not in the map.

If a subclass of `dict` defines a method `__missing__()` and *key* is not present, the `d[key]` operation calls that method with the key *key* as argument. The `d[key]` operation then returns or raises whatever is returned or raised by the `__missing__(key)` call. No other operations or methods invoke `__missing__()`. If `__missing__()` is not defined, `KeyError` is raised. `__missing__()` must be a method; it cannot be an instance variable:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

The example above shows part of the implementation of `collections.Counter`. A different `__missing__` method is used by `collections.defaultdict`.

`d[key] = value`

Set `d[key]` to *value*.

`del d[key]`

Remove `d[key]` from *d*. Raises a `KeyError` if *key* is not in the map.

`key in d`

Return `True` if *d* has a key *key*, else `False`.

`key not in d`

Equivalent to `not key in d`.

`iter(d)`

Return an iterator over the keys of the dictionary. This is a shortcut for `iter(d.keys())`.

Data Types - Mapping: Dictionary Methods

`dict.fromkeys(seq[, value])`

Create a new dictionary with keys from `seq` and values set to `value`.

`fromkeys()` is a class method that returns a new dictionary. `value` defaults to `None`.

`dict.get(key[, default])`

Return the value for `key` if `key` is in the dictionary, else `default`. If `default` is not given, it defaults to `None`, so that this method never raises a `KeyError`.

`dict.items()`

Return a new view of the dictionary's items ((`key`, `value`) pairs). See the [documentation of view objects](#).

`dict.keys()`

Return a new view of the dictionary's keys. See the [documentation of view objects](#).

`dict.pop(key[, default])`

If `key` is in the dictionary, remove it and return its value, else return `default`. If `default` is not given and `key` is not in the dictionary, a `KeyError` is raised.

`dict.popitem()`

Remove and return an arbitrary (`key`, `value`) pair from the dictionary.

`popitem()` is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling `popitem()` raises a `KeyError`.

`dict.setdefault(key[, default])`

If `key` is in the dictionary, return its value. If not, insert `key` with a value of `default` and return `default`. `default` defaults to `None`.

`dict.update([other])`

Update the dictionary with the key/value pairs from `other`, overwriting existing keys. Return `None`.

`update()` accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: `d.update(red=1, blue=2)`.

`dict.values()`

Return a new view of the dictionary's values. See the [documentation of view objects](#).

Data Types - Mapping: Dictionary

```
d1 = {'WR1': 'Odell Beckham Jr.',  
      'WR2': 'Brandon Marshall',  
      'WR3': 'Sterling Shepard',  
      'WR4': 'Roger Lewis'}  
  
d2 = {'WR2': 'Brandon Marshall',  
      'WR1': 'Odell Beckham Jr.',  
      'WR4': 'Roger Lewis',  
      'WR3': 'Sterling Shepard'}  
  
print(d1 == d2)           # Output: True  
print(len(d1))           # Output: 4  
  
print(1 in d1)            # Output: False  
print(d1['WR1'])          # Output: Odell Beckham Jr.
```

Data Types - Mapping: Dictionary

```
receivers = d1
pass_catchers = receivers
pass_catchers['TE1'] = 'Evan Engram'
print(receivers)
print(pass_catchers)
```

dictionary.py* 15:21

LF UTF-8 Python 0 files

```
[slr:Desktop sammishra$ python3 dictionary.py
{'TE1': 'Evan Engram', 'WR1': 'Odell Beckham Jr.', 'WR2': 'Brandon Marshall', 'WR4': 'Roger Lewis', 'WR3': 'Sterling Shepard'}
{'TE1': 'Evan Engram', 'WR1': 'Odell Beckham Jr.', 'WR2': 'Brandon Marshall', 'WR4': 'Roger Lewis', 'WR3': 'Sterling Shepard'}
slr:Desktop sammishra$ ]
```

```
receivers = d1.copy()
pass_catchers = receivers.copy()
pass_catchers['TE1'] = 'Evan Engram'
print(receivers)
print(pass_catchers)
```

dictionary.py 12:33

LF UTF-8 Python 0 files

```
{'WR1': 'Odell Beckham Jr.', 'WR4': 'Roger Lewis', 'WR2': 'Brandon Marshall', 'WR3': 'Sterling Shepard'}
{'TE1': 'Evan Engram', 'WR1': 'Odell Beckham Jr.', 'WR4': 'Roger Lewis', 'WR2': 'Brandon Marshall', 'WR3': 'Sterling Shepard'}
slr:Desktop sammishra$ ]
```

Data Types - Mapping: Dictionary

```
print(d1.items())  
print(d1.keys())  
print(d1.values())  
  
print(d1.get('WR1'))  
print(d1.get('LT'))
```

```
slr:Desktop sammishra$ python3 dictionary.py  
dict_items([('WR3', 'Sterling Shepard'), ('WR2', 'Brandon Marshall'), ('WR4', 'Roger Lewis'), ('WR1', 'Odell Beckham Jr.')])  
dict_keys(['WR3', 'WR2', 'WR4', 'WR1'])  
dict_values(['Sterling Shepard', 'Brandon Marshall', 'Roger Lewis', 'Odell Beckham Jr.'])  
Odell Beckham Jr.  
None  
slr:Desktop sammishra$ |
```

Objectives of this Lesson

1. Set up an Integrated Development Environment (IDE)
2. Data Type - Mapping: dictionaries
3. Functions
4. Variable Scope
5. File I/O
6. Programming with Functions

Functions

Programming languages support decomposing problems in several different ways:

- Most programming languages are **procedural**:
 - programs are lists of instructions that tell the computer what to do with the program's input. C, Pascal, and even Unix shells are procedural languages.
- In **declarative** languages:
 - you write a specification that describes the problem to be solved, and the language implementation figures out how to perform the computation efficiently. SQL is the declarative language you're most likely to be familiar with; a SQL query describes the data set you want to retrieve, and the SQL engine decides whether to scan tables or use indexes, which subclauses should be performed first, etc.
- **Functional** programming:
 - decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don't have any internal state that affects the output produced for a given input. Well-known functional Haskell.

In a functional program, input flows through a set of functions.
Each function operates on its input and produces some output.

Functions

A function is a block of organized, reusable code.

Functions provide better modularity for your application and a high degree of code reusing.

Python gives you many built-in functions but you can also create your own functions. These functions are called *user-defined functions*.

Built-in Functions				
<code>abs()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>all()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>any()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>ascii()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bin()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>bool()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	
<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>	

Functions - Definition

Here are simple rules to define a function in Python:

- Function blocks begin with the keyword **def** followed by the **function name** and **parentheses ()**.
- Any **input parameters** or **arguments** should be placed within these parentheses.
 - You can also define parameters inside these parentheses.
- The code block within every function starts with a **colon :** and is **indented**.
- The statement **return** exits a function, optionally passing back an expression to the caller.
 - A return statement with no arguments is the same as **return None**.

Functions - Definition

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword **def** introduces a **function definition**.

It must be followed by the **function name** and the **parenthesized list of formal parameters**.

The statements that form the **body of the function** start at the next line, and **must be indented**.

The function definition **DOES NOT** execute the function body; this gets executed only when the function is **called**.

Functions - Definition

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring; it's good practice to include docstrings in code that you write, so make a habit of it.

Even functions without a **return** statement do return a value, `None`.

Functions - Return Statement

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

return leaves the current function call with the expression list (or `None`) as return value.

Functions - Arguments

argument

A value passed to a function when calling the function. There are two kinds of argument:

- **keyword argument:** an argument preceded by an **identifier** (e.g. `name=`) in a function call or passed as a value in a **dictionary preceded by `**`**.
 - For example, `3` and `5` are both keyword arguments in the following calls to `complex()`:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```
- **positional argument:** an argument that is not a keyword argument. Positional arguments can appear at the **beginning** of an argument list and/or be passed as elements of an **iterable preceded by `*`**.
 - For example, `3` and `5` are both positional arguments in the following calls:

```
complex(3, 5)
complex(*(3, 5))
```

Arguments are assigned to the named local variables in a function body.

Functions - Arguments

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

Default Argument Values

```
def ask_ok(prompt, retries=4, reminder='Please try again!):
```

Keyword Arguments

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue):
```

Arbitrary Argument Lists

```
def write_multiple_items(file, separator, *args):
```

Functions - Arguments

Default Argument Values

When one or more parameters have the form *parameter* = *expression*, the function is said to have “default parameter values.”

For a parameter with a default value, the corresponding argument may be omitted from a call, in which case the parameter’s default value is substituted.

If a parameter has a default value, all following parameters must also have a default value.

Functions - Arguments

Default Argument Values

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

Functions - Arguments

Default Argument Values

Default parameter values are evaluated from left to right when the function definition is executed.

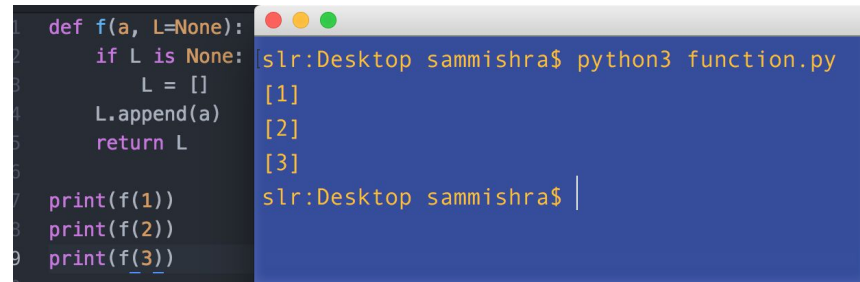
The default value is evaluated only once.

This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes.

```
def f(a, L=[]):  
    L.append(a)  
    return L
```

```
print(f(1))  
print(f(2))  
print(f(3))
```

```
[1]  
[1, 2]  
[1, 2, 3]
```



The screenshot shows a code editor with a Python function definition and its execution in a terminal. The function `f(a, L=None)` initializes a list `L` if it is `None`, appends the argument `a`, and returns it. The terminal shows the command `python3 function.py` being run, which produces the output `[1]`, `[2]`, and `[3]` on separate lines, corresponding to the three `print` statements in the script.

```
def f(a, L=None):  
    if L is None:  
        L = []  
    L.append(a)  
    return L  
  
print(f(1))  
print(f(2))  
print(f(3))
```

slr:Desktop sammishra\$ python3 function.py
[1]
[2]
[3]
slr:Desktop sammishra\$

Functions - Arguments

Keyword Arguments

The following function accepts one required argument (voltage) and three optional arguments (state, action, and type):

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):  
    print("-- This parrot wouldn't", action, end=' ')  
    print("if you put", voltage, "volts through it.")  
    print("-- Lovely plumage, the", type)  
    print("-- It's", state, "!")
```

This function can be called in any of the following ways:

```
parrot(1000) # 1 positional argument  
parrot(voltage=1000) # 1 keyword argument  
parrot(voltage=1000000, action='VOOOOOM') # 2 keyword arguments  
parrot(action='VOOOOOM', voltage=1000000) # 2 keyword arguments  
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments  
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

Functions - Arguments

Keyword Arguments

The following function accepts one required argument (voltage) and three optional arguments (state, action, and type):

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):  
    print("-- This parrot wouldn't", action, end=' ')  
    print("if you put", voltage, "volts through it.")  
    print("-- Lovely plumage, the", type)  
    print("-- It's", state, "!")
```

All the following calls would be invalid:

```
parrot()           # required argument missing  
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument  
parrot(110, voltage=220)     # duplicate value for the same argument  
parrot(actor='John Cleese')  # unknown keyword argument
```

Functions - Arguments

Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments.

- Formal parameter of the form `**name` - receives a dictionary containing all keyword arguments except for those corresponding to a formal parameter.
- Formal parameter of the form `*name` - receives a tuple containing the positional arguments beyond the formal parameter list.

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Objectives of this Lesson

1. Set up an Integrated Development Environment (IDE)
2. Data Type - Mapping: dictionaries
3. Functions
4. Variable Scope
5. File I/O
6. Programming with Functions

Variable Scope

Not all variables are accessible from all parts of our program, and not all variables exist for the same amount of time.

Where a variable is accessible and how long it exists depend on how it is defined.

We call the part of a program where a variable is accessible its **scope**.

There are two basic scopes of variables in Python

- **Global variables**
- **Local variables**

Variables that are defined **inside** a function body have a **local** scope, and those defined **outside** have a **global** scope.

Variable Scope

A variable which is defined in the main body of a file is called a *global* variable. It will be visible throughout the file, and also inside any file which imports that file.

A variable which is defined inside a function is *local* to that function. It is accessible from the point at which it is defined until the end of the function. The parameter names in the function definition behave like local variables. When we use the assignment operator (=) inside a function, it creates a new local variable.

Variable Scope

```
# This is a global variable
a = 0

if a == 0:
    # This is still a global variable
    b = 1

def my_function(c):
    # this is a local variable
    d = 3
    print(c)
    print(d)

# Now we call the function, passing the value 7 as the first and only parameter
my_function(7)

# a and b still exist
print(a)
print(b)

# c and d don't exist anymore -- these statements will give us name errors!
print(c)
print(d)
```

Objectives of this Lesson

1. Set up an Integrated Development Environment (IDE)
2. Data Type - Mapping: dictionaries
3. Functions
4. Variable Scope
5. File I/O
6. Programming with Functions

File I/O

Before you can read or write a file, you have to open it using Python's built-in `open()` function.

This function creates a **file** object, which would be utilized to call other support methods associated with it. It is most commonly used with two arguments: `open(filename, mode)`.

filename is a path-like object giving the pathname (absolute or relative to the current working directory) of the file to be opened.

mode is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode.

Character	Meaning
<code>'r'</code>	open for reading (default)
<code>'w'</code>	open for writing, truncating the file first
<code>'x'</code>	open for exclusive creation, failing if the file already exists
<code>'a'</code>	open for writing, appending to the end of the file if it exists
<code>'b'</code>	binary mode
<code>'t'</code>	text mode (default)
<code>'+'</code>	open a disk file for updating (reading and writing)
<code>'U'</code>	universal newlines mode (deprecated)

File I/O

To read a file's contents, call `fileobject.read(size)`, which reads some quantity of data and returns it as a string .

`size` is an optional numeric argument. When `size` is omitted or negative, the entire contents of the file will be read and returned.

`fileobject.readline()` reads a single line from the file.

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for line in fileobject:
...     print(line, end="")
...
This is the first line of the file.
Second line of the file
```

`fileobject.write(string)` writes the contents of `string` to the file, returning the number of characters written.

File I/O

Call `f.close()` to close the file and free up any system resources used by it.

If you don't explicitly close a file, Python's garbage collector will eventually destroy the object and close the open file for you, but the file may stay open for a while.

It is good practice to use the `with` keyword when dealing with file objects.

The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point.

```
>>> with open('workfile') as f:
...     read_data = f.read()
>>> f.closed
True
```

Objectives of this Lesson

1. Set up an Integrated Development Environment (IDE)
2. Data Type - Mapping: dictionaries
3. Functions
4. Variable Scope
5. File I/O
6. Programming with Functions

Programming

- Wrap lines so that they don't exceed 79 characters.
- This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing constructs:
`a = f(1, 2) + g(3, 4).`
- Name your classes and functions consistently;
 - a. the convention is to use `CamelCase` for classes and
 - b. `lower_case_with_underscores` for functions and methods.

Palindromes

A **palindrome** is a word which reads the same backward as forward, such as *madam* or *racecar*.

Task: Given a word as input determine if it is a palindrome. Return True if the word is a palindrome, False if it is not.

Palindromes

```
palindromes.py
1 def get_word():
2     '''Get the word to be checked from user input'''
3
4     word = input("Enter A Word: ")
5     return word
6
7 def check_word(word):
8     '''Check if the word is a palindrome'''
9
10    length = len(word)
11    for i in range(length//2):
12        left = i
13        right = -(i+1)
14        if word[left] != word[right]:
15            return False
16        return True
17
18 def main():
19     print("This script checks if a word is a Palindrome.")
20     word = get_word()
21     print(check_word(word))
22
23 main()
```

Anagrams

An **anagram** is the result of rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters exactly once; for example, the word anagram can be rearranged into "nag a ram".

Task: Given two words as input determine if they are an anagram of one-another. Return True if the word is an anagram, False if it is not.

Anagrams

```
9 def check_words(word1, word2):
10     '''Check if two words are anagrams of one-another'''
11
12     if len(word1) != len(word2):
13         return False
14
15     dict1 = {}
16     dict2 = {}
17
18     for i in word1:
19         if i not in dict1:
20             dict1[i] = 0
21         else:
22             dict1[i] += 1
23
24     for i in word2:
25         if i not in dict2:
26             dict2[i] = 0
27         else:
28             dict2[i] += 1
29
30     if dict1 == dict2:
31         return True
32     else:
33         return False
```

```
1 def get_words():
2     '''Get the words to be checked from user input'''
3
4     word1 = input("Enter A Word: ")
5     word2 = input("Enter A Word: ")
6
7     return word1, word2
```

```
def main():
    print('This script checks if two words are anagrams of each other.')
    word1, word2 = get_words()
    print(check_words(word1, word2))

main()
```

Fibonacci

the **Fibonacci numbers** are the numbers characterized by the fact that every number after the first two is the sum of the two preceding ones:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

Task: Given an integer as input, determine the fibonacci sequence of that integer. Use recursion (see following slides).

i.e.	fib(1)	returns 0
	fib(5)	returns 0,1,1,2,3
	fib(13)	returns 0,1,1,2,3,5,8,13,34,55,89,144

Fibonacci

Recursion is a way of programming or coding a problem, in which a function calls itself one or more times in its body.

Termination condition:

A recursive function has to terminate to be used in a program. A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case.

A base case is a case, where the problem can be solved without further recursion.

A recursion can lead to an infinite loop, if the base case is not met in the calls.

Example:

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1$$

Replacing the calculated values gives us the following expression

$$4! = 4 * 3 * 2 * 1$$

Fibonacci

```
def factorial(n):  
    print("factorial has been called with n = " + str(n))  
    if n == 1:  
        return 1  
    else:  
        res = n * factorial(n-1)  
        print("intermediate result for ", n, " * factorial(", n-1, "): ", res)  
        return res
```

```
print(factorial(5))
```

Output

```
factorial has been called with n = 5  
factorial has been called with n = 4  
factorial has been called with n = 3  
factorial has been called with n = 2  
factorial has been called with n = 1  
intermediate result for 2 * factorial( 1 ): 2  
intermediate result for 3 * factorial( 2 ): 6  
intermediate result for 4 * factorial( 3 ): 24  
intermediate result for 5 * factorial( 4 ): 120  
120
```

Fibonacci

```
1  def fib(n):
2      if n == 0:
3          return 0
4      elif n == 1:
5          return 1
6      else:
7          return fib(n-1) + fib(n-2)
8
9  def fib_seq(n):
10     for i in range(n):
11         if i == n-1:
12             print(fib(i))
13         else:
14             print(fib(i), end=', ')
15
16  def main():
17     n = int(input('Enter A Number: '))
18     fib_seq(n)
19
20  main()
```