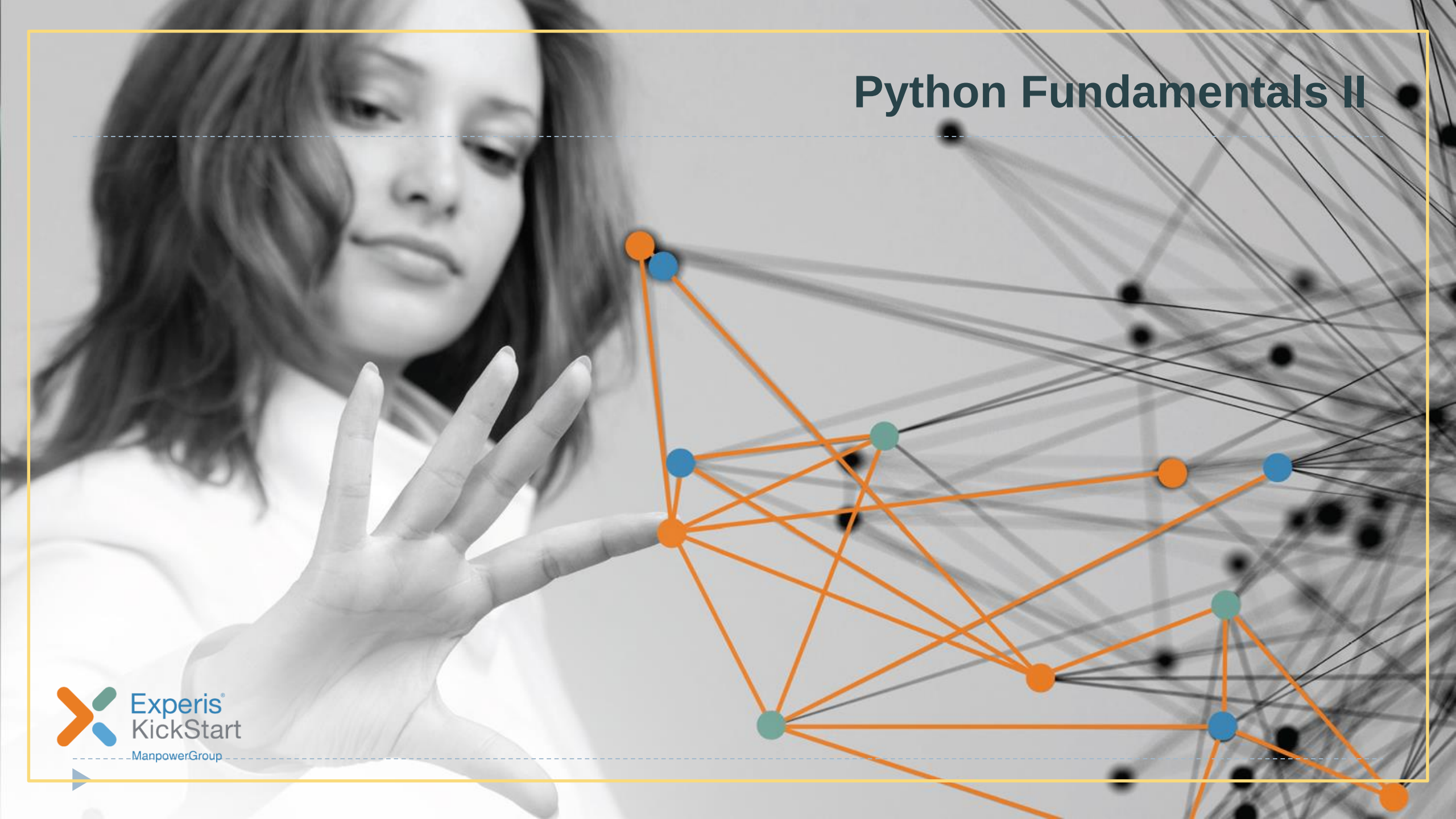
Python Fundamentals II



Experis® KickStart
ManpowerGroup

Functions

Experis KickStart ManpowerGroup

- A function is a named set of statements, which perform a specific task, and can be run more than once in a program
  - So far, we've been using many of Python's built-in functions such as input(), print(), list(), etc.
- There are two main advantages to using functions:
  - They enable code to be reused without having to be replicated in different parts of the program
  - They enable complex tasks to be broken into separate procedures, each implemented by its own function, thus making the code much easier to maintain and understand

Defining and Calling Functions

Experis KickStart ManpowerGroup

- The syntax for defining a function is:

```
def function_name(param1, param2, …):
    <indented statement 1>
    <indented statement 2>
    ...
    <indented statement n>
    <return statement>
```

- A function consists of two parts: a header and a body
- The function header starts with the **def** keyword, followed by name of the function, its arguments and ends with a colon (:)
- The function body contains statements which define what the function does
  - All the statements in the body of the function must be equally indented
  - If at any point during the execution of this statement block a **return statement** is encountered, the specified values are returned to the caller

Defining and Calling Functions

Experis KickStart ManpowerGroup

- For example, the following function gets a number and returns it squared:

```
def square(x):
    x_squared = x ** 2
    return x_squared
```

- Once defined, the function can be called any number of times
- The syntax of calling a function is: `function_name(arg1, arg2, arg3, ..., argN)`
- The arguments passed at the function invocation (**actual parameters**), are bound to the names of the parameters defined in the function header (**formal parameters**)

```
num = 2
num_squared = square(num)
print(num, "squared is", num_squared)
```

```
2 squared is 4
```

```
print("8 squared is", square(8))
```

```
8 squared is 64
```

Defining and Calling Functions

- Function definitions can appear anywhere in a Python program, but a function cannot be referenced or called before it is defined:

```
print("8 squared is", square(8))
def square(x):
    x_squared = x ** 2
    return x_squared
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-8e84aa8742da> in <module>()
----> 1 print("8 squared is", square(8))
      2 def square(x):
      3     x_squared = x ** 2
      4     return x_squared

NameError: name 'square' is not defined
```

Returning a Value

- It is not necessary for a function to explicitly return any object
- If the function body doesn't have any return statement, or if the function falls off the end of its intended block without encountering a return statement, then Python's special value None is returned

```
def add(num1, num2):
    print("Sum is", num1 + num2)
```

```
result = add(10, 20)
print(result)
```

```
Sum is 30
None
```

Returning a Value

- To return two or more values from a function, pack them into a tuple
- For example, the following program defines a function to return both roots of the quadratic equation $ax^2 + bx + c$ (assuming it has two real roots):

```
import math

def solve_quadratic(a, b, c):
    d = b**2 - 4*a*c;
    r1 = (-b + math.sqrt(d)) / (2*a)
    r2 = (-b - math.sqrt(d)) / (2*a)
    return r1, r2
```

```
solve_quadratic(1, -1, -6)
```

```
(3.0, -2.0)
```

Functions as Objects

- In Python, functions are “first class” objects: they can have variables assigned to them, they be passed as arguments and returned from other functions
- A function is given a name when it is defined, but that name can be reassigned to refer to a different object if desired (don’t do this unless you mean to!)
  - That’s the reason why you should not define variables such as range, list, str, etc.
- The following example associates an existing function with a new variable identifier:

```
import math
cosine = math.cos
id(math.cos)
```

```
2683242898056
```

```
id(cosine)
```

```
2683242898056
```

```
cosine(0)
```

```
1.0
```

Naming Functions

Experis KickStart ManpowerGroup

- A function name should clearly describe what the function does
- When we see a function call in the code, a good name instantly gives us an understanding what it does and returns
- A function is an action, thus it is a widespread practice to start a function with a verbal prefix which vaguely describes the action
- For instance, functions starting with…
  - "show…" – usually show something
  - "get…" – return a value
  - "calc…" – calculate something
  - "create…" – create something
  - "check…" – check something and return a boolean

Built-in Functions

- The Python interpreter has a number of functions and types built into it that are always available
- Be careful not to name your functions with the same name as a built-in function

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

One Function – One Action

Experis KickStart ManpowerGroup

- Functions should be short and do exactly one thing
- Two independent actions usually deserve two functions, even if they are usually called together (in that case we can make a 3rd function that calls those two)
- A separate function is not only easier to test and debug – its very existence is a great comment!
- A few examples of breaking this rule:
  - get_age – would be bad if it also prints the age (should only get)
  - check_permission – would be bad if displays the access granted/denied message (should only perform the check and return the result)

DocStrings

- Documentation strings (docstrings) are used to describe what a function does
- The docstring must be the first indented statement in the function or class
- Triple quotes are used while writing docstring
- A function docstring is a string literal that occurs as the first statement of the function definition. It should be written as a triple-quoted string on a single line if the function is simple, or on multiple lines with an initial one-line summary for more detailed descriptions of complex functions

```
def solve_quadratic(a, b, c):
    """Return the roots of ax^2 + bx + c."""
    d = b**2 - 4*a*c;
    r1 = (-b + math.sqrt(d)) / (2*a)
    r2 = (-b - math.sqrt(d)) / (2*a)
    return r1, r2
```

DocStrings

- The docstring becomes the special __doc__ attribute of the function:

```
roots.__doc__
```

```
'Return the roots of ax^2 + bx + c.'
```

- From an interactive shell, typing **help**(function_name) provides more detailed information concerning the function, including this docstring:

```
help(solve_quadratic)
```

```
Help on function solve_quadratic in module __main__:

solve_quadratic(a, b, c)
    Return the roots of ax^2 + bx + c.
```

DocStrings

- We will use Google coding style for writing docstrings:

```
def func(param1, param2):
    """This is an example of Google style.

    Args:
        param1: This is the first param.
        param2: This is a second param.

    Returns:
        This is a description of what is returned.

    Raises:
        KeyError: Raises an exception.
    """
```

```
def square_root(n):
    """Calculate the square root of a number.

    Args:
        n: the number to get the square root of.

    Returns:
        the square root of n.

    Raises:
        TypeError: if n is not a number.
        ValueError: if n is negative.
    """
```

DocStrings

Experis KickStart ManpowerGroup

- Another example for a well-documented function:

```
def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
    """Fetches rows from a Bigtable.

    Retrieves rows pertaining to the given keys from the Table instance represented by big_table.
    Silly things may happen if other_silly_variable is not None.

    Args:
        big_table: An open Bigtable Table instance.
        keys: A sequence of strings representing the key of each table row to fetch.
        other_silly_variable: Another optional variable, that has a much longer name than
            the other args, and which does nothing.

    Returns:
        A dict mapping keys to the corresponding table row data fetched.
        Each row is represented as a tuple of strings. For example:

        {'Serak': ('Rigel VII', 'Preparer'),
         'Zim': ('Irk', 'Invader'),
         'Lrrr': ('Omicron Persei 8', 'Emperor')}

        If a key from the keys argument is missing from the dictionary,
        then that row was not found in the table.

    Raises:
        IOError: An error occurred accessing the bigtable.Table object.
    """
```

Exercise (1)

- Write a function that returns the greatest common divisor (GCD) of two numbers
- The greatest common divisor of two numbers is the largest positive integer that perfectly divides the two given numbers. For example, the GCD of 12 and 15 is 3.
- Use the Euclidean algorithm for finding the GCD:
  - Divide the greater number by the smaller and take the remainder
  - Now, divide the smaller by this remainder
  - Repeat until the remainder is 0
- For example:

$$\begin{aligned}\gcd(1071, 462) &= \gcd(462, 1071 \bmod 462)\\ &= \gcd(462, 147)\\ &= \gcd(147, 462 \bmod 147)\\ &= \gcd(147, 21)\\ &= \gcd(21, 147 \bmod 21)\\ &= \gcd(21, 0)\\ &= 21\end{aligned}$$

Exercise (2)

Experis KickStart ManpowerGroup

- A **perfect number** is an integer that is equal to the sum of its proper divisors (the sum of its divisors excluding the number itself)
  - The first perfect number is 6, since 6 = 1 + 2 + 3
  - The next perfect number is 28, since 28 = 1 + 2 + 4 + 7 + 14
- There are many interesting mathematical open problems regarding perfect numbers
  - For example, it is not known whether there are any odd perfect numbers, nor whether infinitely many perfect numbers exist
- Write a function is_perfect(n) that returns whether n is a perfect number or not
- Write a function find_perfect_numbers(n) that returns a list of all the perfect numbers which are less than or equal to n
- Use the last function to print all perfect numbers less than 10,000
- Use docstrings to document all your functions

Exercise (3)

Experis KickStart ManpowerGroup

- Write a function rank(scores) that gets a list of scores, and produces a list associating each score with a rank (starting with 1 for the highest score)
- Equal scores should have the same rank
- Sample usage of the function:

```
scores = [87, 75, 75, 50, 32, 32]
score_ranks = rank(scores)
score_ranks
```

```
[1, 2, 2, 4, 5, 5]
```

Exercise (4)

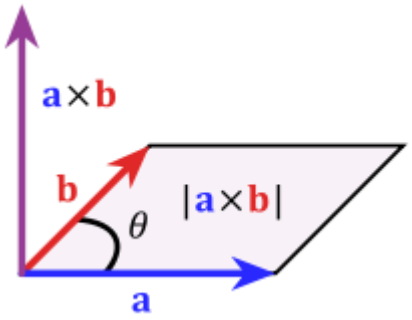
- Write two functions which, given two lists of length 3 representing three dimensional vectors **a** and **b**, calculate the dot product, **a** · **b**, and the vector (cross) product, **a** × **b**
- **Reminder:** the dot product of two vectors **a** and **b** is defined as:

$$\mathbf{a}\cdot\mathbf{b}=\sum_{i=1}^{n}a_i b_i = \|\mathbf{a}\|\|\mathbf{b}\|\cos(\theta)$$

  - where θ is the angle between **a** and **b**
- The cross product of two vectors **a** and **b** is defined only for 3-dimensional vectors:

$$\mathbf{a}\times\mathbf{b}=(a_2b_3-a_3b_2,a_3b_1-a_1b_3,a_1b_2-a_2b_1)=\|\mathbf{a}\|\|\mathbf{b}\|\sin(\theta)\mathbf{n}$$

  - where **n** is a unit vector perpendicular to the plane containing **a** and **b** in the direction given by the right-hand rule
- The cross product is a vector that is perpendicular to both **a** and **b**



Exercise (5)

Experis KickStart ManpowerGroup

- A DNA sequence encodes each amino acid making up a protein as a three nucleotide sequence called a *codon*
- For example, the sequence fragment AGTCTTATATCT contains the codons (AGT, CTT, ATA, TCT) if read from the first position (*"frame"*)
  - If read in the second frame it yields the codons (GTC, TTA, TAT), and in the third (TCT, TAT, ATC)
- Write a function that extracts the codons into a list of 3-letter strings given a DNA sequence and a frame as an integer value (0, 1 or 2)

Variable Scope

Experis KickStart ManpowerGroup

- The scope of a variable refers to the part of the program where it can be accessed
- **Local variables** are variables created inside a function
  - Local variables can only be accessed inside the body of the function in which it is defined
  - Local variables are subject to garbage collection as soon as the function ends
    - As a result, trying to access a local variable outside it's scope will result in an error
- **Global variables** are defined outside of any function
  - The scope of a global variable starts from the point where it is defined and continues until the end of the program
- Some notes on global variables:
  - Use global variables only when it’s important that these variables are accessible from anywhere
  - Modern code has few or no globals
  - Most variables reside in their functions

Variable Scope

Experis KickStart ManpowerGroup

- Example:

```
global_var = 5 # a global variable

def func():
    local_var = 10 # a local variable, only avaiable inside func()
    print("Inside func(): local_var =", local_var)
    print("Inside func(): global_var =", global_var) # accessing a global variable inside a function

func()
print("Outside func(): global_var =", global_var)
```

```
Inside func(): local_var = 10
Inside func(): global_var = 5
Outside func(): global_var = 5
```

- The global variable can be defined after the function is defined, but must be before the function is called:

```
def func():
    a = 5
    print(a, b)
b = 6
func()
```

```
5 6
```

Variable Scope

- If we have a local and a global variable with the same name, the local variable *shadows* the global one:

```
user_name = "John"
def show_message():
    user_name = "Bob" # shadowing
    print("Hello,", user_name)

show_message()
print(user_name)
```

```
Hello, Bob
John
```

The local variable user_name exists only within the body of the function. It disappears after the function exits, and doesn't overwrite the global user_name.

- Python's rules for resolving scope can be summarized as **LEGB**:

  Local scope -> Enclosing scope -> Global scope –> Built-ins
- Thus, if you happen to give a variable the same name as a built-in function (such as range or len), then that name resolves to your variable
  - It is therefore generally not a good idea to name your variables after built-ins

The global keyword

- A function cannot *modify* variables defined outside its local scope ("rebind" them to new objects)
  - Since an assignment statement inside a function is interpreted as defining a new local variable

```
def func1():
    print(x)   # OK, providing x is defined in global or enclosing scope
```

```
def func2():
    x += 1   # Not OK: can't modify x if it isn't local
```

```
x = 4
func1()
```

```
4
```

```
func2()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-139-1159c30513e1> in <module>()
----> 1 func2()

<ipython-input-137-aeafdc746753> in func2()
      1 def func2():
----> 2     x += 1   # Not OK: can't modify x if it isn't local

UnboundLocalError: local variable 'x' referenced before assignment
```

The global keyword

- The **global** keyword allows you to modify a global variable within a local scope

```
def func2():
    global x
    x += 1   # OK now - Python knows we mean x in global scope
```

```
x = 4
func2()   # No error
x
```

```
5
```

- You should think carefully whether it is really necessary to use this technique
  - Would it be better to pass x as an argument and return its updated value from the function?
  - Especially in longer programs, variable names in one scope that change value within functions lead to confusing code, behavior that is hard to predict and tricky bugs

Nested Functions

- **Nested functions** are functions defined within other functions
  - An arbitrary level of nesting is possible
- We can use nested functions to organize our code, like this:

```
def say_hi_bye(first_name, last_name):
    def get_full_name():
        return first_name + " " + last_name

    print("Hi,", get_full_name())
    print("Bye,", get_full_name())
```

```
say_hi_bye("Adam", "Smith")
```

```
Hi, Adam Smith
Bye, Adam Smith
```

NonLocal Variables

- A nested function can read variables declared in its enclosing scope
  - But cannot assign new values to that variables
- Similarly to global, a variable can be declared in the nested function as **nonlocal**
- Once this is done, the nested function can assign a new value to that variable and that modification is going to be seen outside of the nested function:

```
def outer():
    x = 5

    def nested():
        nonlocal x
        x += 1

    nested()
    print(x)

outer()
```

```
6
```

Closures

- A **closure** is a nested function that is returned by the outer function, and can access the outer variables even after the outer function has finished its execution
- Closures can avoid the use of global values and provide some form of data hiding

```
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier
```

```
times3 = make_multiplier_of(3)
times5 = make_multiplier_of(5)
```

```
print(times3(9))
```

```
27
```

```
print(times5(3))
```

```
15
```

  - Note that even when the outer function has finished its execution, the closure function multiplier(x) returned by it can refer to the variable of the outer function (n in this case)

Closures

- All closure functions have a **__closure__** attribute that returns a tuple of cell objects
- The cell object has an attribute cell_contents which stores the closed value:

```
times3.__closure__[0].cell_contents
```

```
3
```

```
times5.__closure__[0].cell_contents
```

```
5
```

Exercise (6)

- Study the following code and predict the result before running it:

```
def outer_func():
    def inner_func():
        a = 9
        print("inside inner_func, a is {} (id={})".format(a, id(a)))
        print("inside inner_func, b is {} (id={})".format(b, id(b)))
        print("inside inner_func, len is {} (id={})".format(len, id(len)))
    len = 2
    print("inside outer_func, a is {} (id={})".format(a, id(a)))
    print("inside outer_func, b is {} (id={})".format(b, id(b)))
    print("inside outer_func, len is {} (id={})".format(len, id(len)))
    inner_func()

a, b = 6, 7
outer_func()
print("in global scope, a is {} (id={})".format(a, id(a)))
print("in global scope, b is {} (id={})".format(b, id(b)))
print("in global scope, len is {} (id={})".format(len, id(len)))
```

Parameter Passing

Experis KickStart ManpowerGroup

- The two common mechanisms for passing arguments to functions are:
  - **Call by Value** – a copy of the actual arguments is passed to the respective formal arguments, hence any change to the values of the formal arguments will not affect the variables in the caller's scope.
  - **Call by Reference** – the location (address) of the actual arguments is passed to the formal arguments, hence any change made to the formal arguments will also reflect in the actual arguments.
- Python uses a mechanism which is known as "**Call by Object**":
  - Recall that everything in Python is object
  - Thus, when a function is called with arguments, the address of the object stored in the argument is passed to the parameter variable
  - If you pass immutable arguments like strings or tuples, the passing acts like call-by-value: the object reference is passed to the function parameters but they can't be changed in the function.
  - If you pass mutable arguments (such as lists), they are also passed by object reference, but they can be changed in place in the function.

Passing Immutable Objects

```
def func1(a):
    print("func1: a = {}, id = {}".format(a, id(a)))
    a = 7  # reassigns local a to the integer 7
    print("func1: a = {}, id = {}".format(a, id(a)))
```

```
a = 3
print("global: a = {}, id = {}".format(a, id(a)))
```

```
global: a = 3, id = 1387949600
```

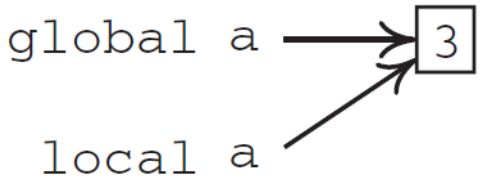
```
func1(a)
```

```
func1: a = 3, id = 1387949600
func1: a = 7, id = 1387949728
```

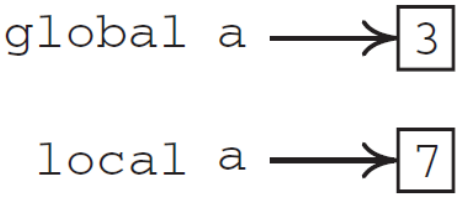
```
print("global: a = {}, id = {}".format(a, id(a)))
```

```
global: a = 3, id = 1387949600
```

Before reassigning the local variable a



After reassigning the local variable a



Passing Mutable Objects

```
def func2(b):
    print("func1: b = {}, id = {}".format(b, id(b)))
    b.append(7)  # add an item to the list
    print("func1: b = {}, id = {}".format(b, id(b)))
```

```
c = [1, 2, 3]
print("global: c = {}, id = {}".format(c, id(c)))
```

```
global: c = [1, 2, 3], id = 2122945792840
```

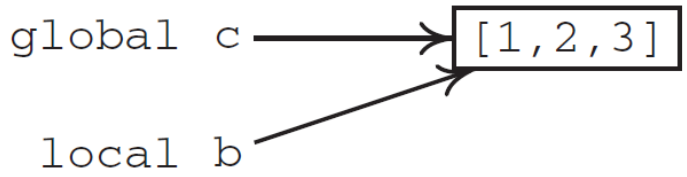
```
func2(c)
```

```
func1: b = [1, 2, 3], id = 2122945792840
func1: b = [1, 2, 3, 7], id = 2122945792840
```

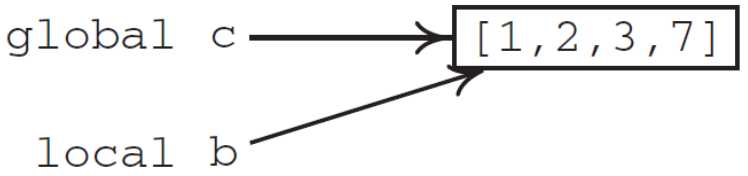
```
print("global: c = {}, id = {}".format(c, id(c)))
```

```
global: c = [1, 2, 3, 7], id = 2122945792840
```

Before appending to the list



After appending to the list



Keyword Arguments

- Arguments to functions can be passed in two ways:
  - **Positional arguments** – the arguments are passed in the same order as their respective parameters in the function header
  - **Keyword arguments** – the arguments are passed in an arbitrary order by setting them explicitly, using the form: kwarg=value
- Here are some different ways to call solve_quadratic() using keyword arguments:

```
solve_quadratic(a=1, c=-6, b=-1)
```

```
(3.0, -2.0)
```

```
solve_quadratic(b=-1, a=1, c=-6)
```

```
(3.0, -2.0)
```

```
solve_quadratic(c=-6, b=-1, a=1)
```

```
(3.0, -2.0)
```

Keyword Arguments

- Keyword arguments are useful in the following scenarios:
  - You want to call arguments by their names to make it more clear what they represent
  - You want to rearrange arguments in a way that makes them most readable
  - You want to leave out arguments that have default values (to be discussed next)
- We can also mix positional and keyword arguments in a function call
- In such case the positional arguments must come before any keyword arguments
  - Otherwise Python won't know to which variable the positional argument corresponds

```
solve_quadratic(1, c=-6, b=-1)   # OK
```

```
(3.0, -2.0)
```

```
solve_quadratic(b=-1, 1, -6)  # Oops: which is a and which is c?
```

```
  File "<ipython-input-58-a11319e0ccd7>", line 1
    solve_quadratic(b=-1, 1, -6)  # Oops: which is a and which is c?
                          ^
SyntaxError: positional argument follows keyword argument
```

Default Arguments

Experis KickStart ManpowerGroup

- Sometimes you want to define a function that takes an *optional argument*: if the caller doesn't provide a value for this argument, a default value is used
- Default arguments are set in the function definition, by using an assignment operator following the parameter name:

```
def log(message, severity="INFO"):
    print("{}: {}".format(severity, message))
```

```
log("File closed")
```

```
INFO: File closed
```

```
log("Cannot open file", "ERROR")
```

```
ERROR: Cannot open file
```

- Parameters with default arguments must be the trailing parameters in the function declaration parameter list

Default Arguments

Experis KickStart ManpowerGroup

- In a function call, when you skip some parameters with default arguments, and want to provide values for the rest, you must use keyword arguments:

```
def log(message, severity="INFO", source="FileSystem"):
    print("[{}] {}: {}".format(source, severity, message))
```

```
log("Table created", source="SQLServer")
```

```
[SQLServer] INFO: Table created
```

Default Arguments

Experis KickStart ManpowerGroup

- Default arguments are assigned when the Python interpreter first encounters the function definition
- Therefore, if a function is defined with an argument defaulting to the value of some variable, subsequently changing that variable *will not change the default:*

```
default_severity="INFO"
def log(message, severity=default_severity, source="FileSystem"):
    print("[{}] {}: {}".format(source, severity, message))
```

```
log("File closed")
```

```
[FileSystem] INFO: File closed
```

```
default_severity="WARNING"
log("File closed")
```

```
[FileSystem] INFO: File closed
```

Mutable Default Arguments

Experis KickStart ManpowerGroup

- This can lead to some unexpected results, particularly for mutable arguments:

```
def append_to(element, alist=[]):
    alist.append(element)
    return alist
```

```
list1 = append_to(12)
print(list1)

list2 = append_to(25)
print(list2)
```

```
[12]
[12, 25]
```

  - A new list is created *once* when the function is defined, and the same list is used in each successive call
  - This means that if you use a mutable default argument and mutate it, you have mutated that object for all future calls to the function as well

Mutable Default Arguments

- If you want to create a new object each time the function is called, you can use a default arg to signal that no argument was provided (**None** is often a good choice):

```
def append_to(element, alist=None):
    if alist is None:
        alist = []
    alist.append(element)
    return alist
```

```
list1 = append_to(12)
print(list1)

list2 = append_to(25)
print(list2)
```

```
[12]
[25]
```

Variable Length Argument Lists

Experis KickStart ManpowerGroup

- In Python you can pass an arbitrary number of arguments to functions
- A single asterisk (*args) is used to pass a *non-keyworded*, variable-length argument list
  - *args will be bound to a tuple containing any positional arguments which don't correspond to any formal parameter
- A double asterisk (**kargs) is used to pass a *keyworded*, variable-length argument list
  - **kargs will be bound to a dictionary containing any keyword arguments which don't correspond to any formal parameters
- The arguments in a function or a function call must appear in the following order:
  - Positional arguments
  - *args
  - Keyword arguments
  - **kwargs

Arbitrary Positional Arguments

- Examples for functions with arbitrary positional arguments:

```
def multiply(*args):
    result = 1
    for num in args:
        result *= num
    return result
```

```
multiply(1, 2, 3, 4, 5)
```

```
120
```

```
multiply()  # args will be an empty tuple
```

```
1
```

```
def test_var_args(f_arg, *args):
    print("first normal arg:", f_arg)
    for arg in args:
        print("another arg through *args:", arg)

test_var_args('yasoob', 'python', 'eggs', 'test')
```

```
first normal arg: yasoob
another arg through *args: python
another arg through *args: eggs
another arg through *args: test
```

- The built-in function print() takes an unlimited number of positional arguments, thus the optional sep, end, file, and flush attributes must be passed as keyword arguments:

```
print("comma", "separated", "word", sep=", ")
```

```
comma, separated, word
```

Arbitrary Keyword Arguments

Experis KickStart ManpowerGroup

- An example for a function with arbitrary keyword arguments:

```
def print_values(**kwargs):
    for key, value in kwargs.items():
        print("The value of {} is {}".format(key, value))

print_values(my_name="thor", your_name="hulk")
```

```
The value of my_name is thor
The value of your_name is hulk
```

Arbitrary Keyword Arguments

- Combining arbitrary keyword arguments with arbitrary positional arguments:

```
def print_receipt(order_id, *food_list, **keywords):
    print("Order #", order_id, sep="")
    print("-" * 30)
    for food in food_list:
        print(food)
    print()
    for kw in keywords:
        print(kw, ": ", keywords[kw], sep="")
```

```
print_receipt(186, "Greek salad", "Tarragon Chicken", "Creamy Burrito",
              Total=81.5,
              Cash=100,
              Change=18.5)
```

```
Order #186
------------------------------
Greek salad
Tarragon Chicken
Creamy Burrito

Total: 81.5
Cash: 100
Change: 18.5
```

Keyword-Only Arguments

- Keyword-only parameters can be defined by including a bare * in the parameter list of the function definition before them
  - This ensures that users of the function cannot accidentally use an option that is controlled by a keyword-only argument
- For example:

```
def func(parameter, *, option1=False, option2=''):
    pass
```

- In this example, option1, and option2 are only specifiable via keyword arguments:

```
func(3, option1=True, option2="Hello world")
```

```
func(3, True, "Hello world")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-36-7e457df80701> in <module>()
----> 1 func(3, True, "Hello world")

TypeError: func() takes 1 positional argument but 3 were given
```

Unpacking Argument Lists

Experis KickStart ManpowerGroup

- * and ** can be used not only in function definitions, but also when calling a function
- For example, when the arguments are already in a list or tuple, but need to be unpacked for a function call requiring separate positional arguments:

```
def f(x, y, z):
    print("x:", x)
    print("y:", y)
    print("z:" , z)
```

```
p = [47, 11, 12]
f(*p)
```

```
x: 47
y: 11
z: 12
```

```
args = [3, 10]
list(range(*args))
```

```
[3, 4, 5, 6, 7, 8, 9]
```

Unpacking Argument Lists

- In the same fashion, dictionaries can deliver keyword arguments with the **-operator:

```
def parrot(voltage, state='a stiff', action='voom'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.", end=' ')
    print("E's", state, "!")

d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
parrot(**d)
```

```
-- This parrot wouldn't VOOM if you put four million volts through it. E's blee
din' demised !
```

Exercise (7)

Experis KickStart ManpowerGroup

- Given the definition of the following six functions:

```
def f1(a, b): print(a, b)
def f2(a, *b): print(a, b)
def f3(a, **b): print(a, b)
def f4(a, *b, **c): print(a, b, c)
def f5(a, b=2, c=3): print(a, b, c)
def f6(a, b=2, *c): print(a, b, c)
```

- What will be the result of the following expressions? Explain and test each result

```
f1(1, 2)
f1(b=2, a=1)
f2(1, 2, 3)
f3(1, x=2, y=3)
f4(1, 2, 3, x=2, y=3)
f5(1)
f5(1, 4)
f6(1)
f6(1, 3, 4)
```

Lambda Functions

Experis KickStart ManpowerGroup

- A lambda function in Python is a type of *anonymous* (unnamed) function
- It is used primarily to write short functions that are a hassle to define as regular functions
- A lambda function has the following syntax:

```
lambda arguments: expression
```

  - Lambda functions can have any number of arguments but only one expression
  - The expression is evaluated and returned
  - The expression cannot contain statements such as loop blocks, conditionals or print statements
- Lambda functions can be used wherever function objects are required

Lambda Functions

- An example for a lambda function that doubles the input value:

```
double = lambda x: x * 2

print(double(5))
```

```
10
```

  - The argument 5 is passed to x and the result of the expression specified in the lambda definition after the colon is passed back to the caller

- To pass more than one argument to a lambda function, pass a tuple (without parentheses):

```
f = lambda x, y: (x + y) ** 2
f(2, 3)
```

```
25
```

Lambda Functions as Arguments

Experis KickStart ManpowerGroup

- Lambda is often used as an argument to other functions that expect a function object
- For example, the built-in sorted() and sort() functions can order lists based on a **key function**, which is called on each element prior to making comparisons
- For example, sorting a list of strings is case sensitive by default:

```
sorted("Nobody expects the Spanish Inquisition".split())
```

```
['Inquisition', 'Nobody', 'Spanish', 'expects', 'the']
```

- We can make the sorting case insensitive, however, by passing each word to the str.lower (or str.upper) method:

```
sorted("Nobody expects the Spanish Inquisition".split(), key=str.lower)
```

```
['expects', 'Inquisition', 'Nobody', 'Spanish', 'the']
```

  - We do not use parentheses here, as in str.lower(), because we are passing the *function itself* to the key argument, not calling it directly

Lambda Functions as Arguments

- It is typical to use lambda expressions to provide simple anonymous functions for other functions
- For example, the built-in function **sorted**(*iterable*, *, *key=None*, *reverse=False*) has an optional parameter **key**, that specifies a function of one argument that is used to extract a comparison key from each list element
- In the following example we use a lambda expression to sort a list of tuples by the second item of each tuple, so the noble gases are sorted by their atomic number:

```
noble_gases = [("argon", 18), ("neon", 10), ("xenon", 54),
               ("krypton", 36), ("helium", 2), ("radon", 86)]
sorted(noble_gases, key=lambda e: e[1])
```

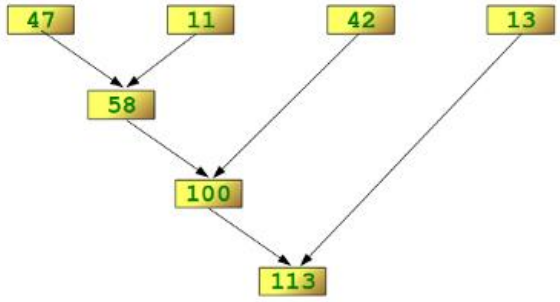
```
[('helium', 2),
 ('neon', 10),
 ('argon', 18),
 ('krypton', 36),
 ('xenon', 54),
 ('radon', 86)]
```

Reducing a List

- The function **reduce**(*func*, *seq*) continually applies the function *func*() to the sequence *seq*, until a single value is returend
- If seq = [s_1, s_2, s_3, ... , s_n], calling reduce(*func*, *seq*) works like this:
  - At first, the first two elements of seq will be applied to func, i.e. func(s_1,s_2)
  - The list on which reduce() works looks now like this: [func(s_1, s_2), s_3, ... , s_n]
  - Next, func() is applied on the previous result and the third element, i.e. func(func(s_1, s_2), s_3)
  - The list looks like this now: [func(func(s_1, s_2), s_3), ... , s_n]
  - Continue like this until just one element is left and return this element as the result of reduce()

```
from functools import reduce
reduce(lambda x, y: x + y, [47, 11, 42, 13])
```

```
113
```



Exercise (8)

Experis KickStart ManpowerGroup

- Define a function **apply**(func, list) that gets a function and a list, and applies the function on every item in the list (modifying the list in place)
- Use apply() to multiply by 5 all the numbers in a given list of numbers
- Use apply() to capitalize all the words in a given list (i.e., change the first letter to uppercase)

Exercise (9)

Experis KickStart ManpowerGroup

- Predict the result of the following program before running it:

```
def func1(f):
    return lambda x: f(x + 1)

func2 = func1(lambda x: x * x)
func2(2)
```

Generators

- A *generator* is a function that returns a *generator iterator* by calling **yield**
- It is a function that behaves like an iterable object
  - i.e., a function that can be used in a for loop and that will yield its values, in turn, on demand
  - This is often more efficient than calculating and storing all of the values that will be iterated over
- A generator function looks just like a regular Python function, but instead of exiting with a return value, it contains a **yield** statement which returns a value each time it is required to by the iteration
- In fact, we've been using generators already because the familiar range() built-in function is a type of generator object (from Python 3)

Generators

Experis KickStart ManpowerGroup

- As a simple example, let's define a generator, **count()**, to count to n:

```
def count(n):
    for i in range(n):
        yield i
```

```
for j in count(5):
    print(j)
```

```
0
1
2
3
4
```

- Note that we can't simply call our generator like a regular function:

```
count(5)
```

```
<generator object count at 0x0000019E19DD6678>
```

- The generator count is expecting to be called as part of a loop and on each iteration it yields its result and stores its state (the value of i) until the loop next calls upon it

Generator Comprehension

- There is a *generator comprehension* syntax similar to list comprehension (use round brackets instead of square brackets):

```
squares = (x**2 for x in range(5))
```

```
for square in squares:
    print(square)
```

```
0
1
4
9
16
```

- However, once we have “exhausted” our generator comprehension, we cannot iterate over it again without redefining it:

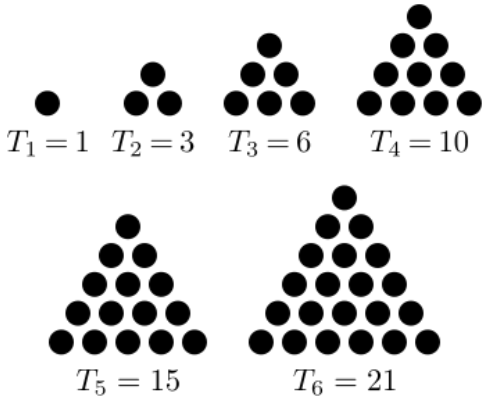
```
for square in squares:
    print(square)
```

- To obtain a list or tuple of a generator’s values, simply pass it to list() or tuple()

Exercise (10)

- Define a generator triangular_numbers(n) that generates the triangular numbers:

$$T_n = \sum_{k=1}^{n} k = 1+2+3+\ldots+n \quad n = 0,1,2,\ldots$$



- Use this function to generate the list of the first 15 triangular numbers

Filtering

- The built-in function **filter**(*func*, *iter*) returns a generator-like object, that generates the elements of an iterable, for which the function *func* returns True
- In the following example, the odd integers less than 10 are generated:

```
list(filter(lambda x: x % 2 == 1, range(10)))
```

```
[1, 3, 5, 7, 9]
```

- This statement is equivalent to the list comprehension:

```
[x for x in range(10) if x % 2 == 1]
```

```
[1, 3, 5, 7, 9]
```

The map() Function

Experis KickStart ManpowerGroup

- The built-in function **map**(*func*, *iter*) returns a generator-like object that applies the function *func* to all the elements of the iterable *iter*, yielding the results one by one
- For example, one way to sum a list of lists is to map the sum built-in to it:

```
mylists = [[1, 2, 3], [10, 20, 30], [25, 75, 100]]
list(map(sum, mylists))
```

```
[6, 60, 200]
```

- This statement is equivalent to the list comprehension:

```
[sum(l) for l in mylists]
```

```
[6, 60, 200]
```

Exercise (11)

- You are given the following sentence:

```
s = ["There", "are", "no", "miracles", "in", "this", "world"]
```

- Print the lengths of all the words in the sentence that contain the letter ‘a’
  - Using map() and filter()
  - Using list comprehension

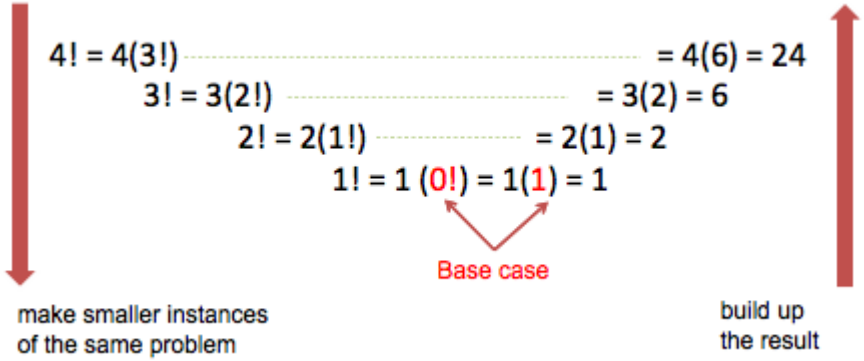
Recursion

- A **recursive function** is a function that calls itself
- Recursion is not always necessary but can often lead to elegant algorithms
- Following is an example of recursive function to find the factorial of an integer
  - The algorithm makes use of the fact that n! = n · (n-1)!

```
def factorial(n):
    """A recursive function to find the factorial of an integer"""
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

```
num = int(input("Enter a number:"))
print("The factorial of", num ,"is", factorial(num))
```

```
Enter a number:5
The factorial of 5 is 120
```



Recursion

Experis KickStart ManpowerGroup

- Advantages of recursion:
  - Recursive functions make the code look clean and elegant
  - A complex task can be broken down into simpler sub-problems using recursion
  - Sequence generation is easier with recursion than using some nested iterations
- Disadvantages of recursion:
  - Sometimes the logic behind recursion is hard to follow through
  - Recursive calls are expensive (inefficient) as they take up a lot of memory and time
  - Recursive functions are hard to debug

Exercise (12)

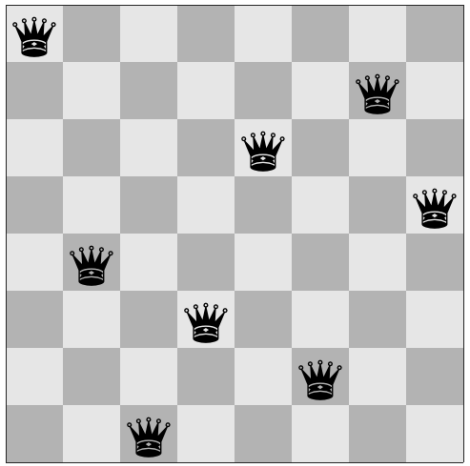
Experis KickStart ManpowerGroup

- Write a function that determines if a string is a palindrome (that is, reads the same backward as forward) **using recursion**
- Some examples of palindromic words are
  redivider, *deified*, *civic*, *radar*, *level*, *rotor*, *kayak*, *reviver*, *racecar*, *redder*, *madam*, and *refer*

Exercise (13)

Experis KickStart ManpowerGroup

- The **N-Queen problem** is the problem of placing N chess queens on an N×N chessboard, so that no two queens attack each other
- Thus, a solution requires that no two queens share the same row, column, or diagonal
- For example, the following is a solution for 8-Queen problem:



- Write a function that gets the board size N, prints the number of possible placements of N queens on the board of size N×N, and all the possible boards

Exercise (13) – Cont.

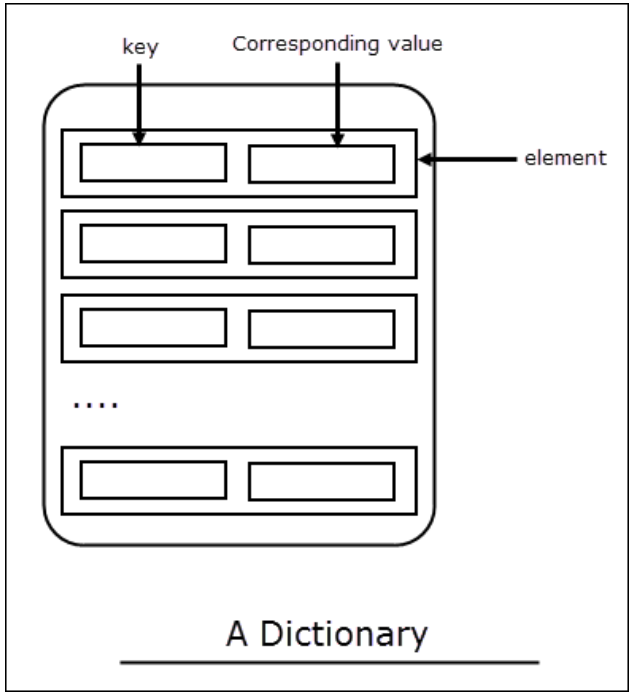
- Example:

```
count = 0
place_queens(6)
print("Number of boards:", count)
```

```
.Q....
...Q..
.....Q
Q.....
..Q...
....Q.

..Q...
.....Q
.Q....
....Q.
Q.....
...Q..

...Q..
Q.....
....Q.
.Q....
.....Q
..Q...

....Q.
..Q...
Q.....
.....Q
...Q..
.Q....

Number of boards: 4
```

Dictionaries

- A **dictionary** is a collection of key-value pairs
  - Also known as “associative array” or “hash” in other languages
- Unlike sequences such as lists and tuples, in which items are indexed by an integer, each item in a dictionary is indexed by a unique key
- The items are stored in the dictionary in no particular order
- The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings or tuples
  - Actually, dictionary keys can be any hashable objects, i.e. objects with a __hash__() method that generates a particular integer from any instance of that object
- Dictionaries themselves are mutable objects



Defining a Dictionary

- A dictionary is defined by giving key: value pairs between braces:

```
variable_name = {
    'key1': value1,
    'key2': value2,
    ...
    'keyN': valueN
}
```

- For example:

```
height = {
    'Burj Khalifa': 829.8,
    'One World Trade Center': 541.3,
    'Tokyo Skytree': 634,
    '432 Park Avenue': 425.5,
    'KVLY-TV Mast': 628.8
}
```

- Although the dictionary keys must be unique, the dictionary values need not be

Defining a Dictionary

- The command **print**() will display the dictionary in the same format (between braces), but in no particular order

```
print(height)
```

```
{'Burj Khalifa': 829.8, 'One World Trade Center': 541.3, 'Tokyo Skytree':
634, '432 Park Avenue': 425.5, 'KVLY-TV Mast': 628.8}
```

- We can also use **dict()** constructor function to define a dictionary
  - The function requires you to pass a sequence of (key, value) pairs
  - If the keys are simple strings (i.e., strings that could be used as variable names), the pairs can also be specified as keyword arguments to the constructor:

```
capitals = dict([('US', 'Washington'), ('France', 'Paris'), ('UK', 'London')])
mass = dict(Mercury=3.301e23, Venus=4.867e24, Earth=5.972e24)
```

- To create an empty dictionary use {}, or dict()

Accessing Values in a Dictionary

- An individual item can be retrieved by indexing it with its key, either as a literal or with a variable equal to the key:

```
height['One World Trade Center']
```

```
541.3
```

```
building = 'Tokyo Skytree'
height[building]
```

```
634
```

- If the specified key doesn't exist, a KeyError exception will be raised:

```
height['The Shard']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-30-3c9dcfb9ffab> in <module>()
----> 1 height['The Shard']

KeyError: 'The Shard'
```

Adding and Modifying Values

Experis KickStart ManpowerGroup

- Items in a dictionary can also be *assigned* by indexing it in this way:

```
height['Empire State Building'] = 381
height
```

```
{'432 Park Avenue': 425.5,
 'Burj Khalifa': 829.8,
 'Empire State Building': 381,
 'KVLY-TV Mast': 628.8,
 'One World Trade Center': 541.3,
 'Tokyo Skytree': 634}
```

- If the key already exists in the dictionary, then its value is updated:

```
height['Empire State Building'] = 443  # with antenna included
height
```

```
{'432 Park Avenue': 425.5,
 'Burj Khalifa': 829.8,
 'Empire State Building': 443,
 'KVLY-TV Mast': 628.8,
 'One World Trade Center': 541.3,
 'Tokyo Skytree': 634}
```

Deleting Elements

- To delete an element from a dictionary, use the **del** statement:

```
del height['KVLY-TV Mast']
height
```

```
{'432 Park Avenue': 425.5,
 'Burj Khalifa': 829.8,
 'Empire State Building': 443,
 'One World Trade Center': 541.3,
 'Tokyo Skytree': 634}
```

- If the key doesn't exist in the dictionary, a KeyError exception is raised
- We can use the built-in **len()** function to get the number of elements in a dictionary:

```
len(height)
```

```
5
```

For Loop Iteration

Experis KickStart ManpowerGroup

- We can use a for loop to iterate over all the keys in the dictionary:

```
for building in height:
    print(building, height[building], sep=': ')
```

```
Burj Khalifa: 829.8
One World Trade Center: 541.3
Tokyo Skytree: 634
432 Park Avenue: 425.5
Empire State Building: 443
```

Membership Operators

- The **in** and **not in** operators can be used to test the existence of a key inside a dictionary:

```
'Tokyo Skytree' in height
```

```
True
```

```
'The Shard' in height
```

```
False
```

```
'The Shard' not in height
```

```
True
```

Comparison Operators

- We can use == and != operators to test whether two dictionaries contain the same elements or not:

```
marks1 = {'Larry': 90, 'Bill': 60}
marks2 = {'Bill': 60, 'Larry': 90}
marks1 == marks2
```

```
True
```

- The other comparison operators such as <, <=, > and >= can't be used with dictionaries because elements in dictionary are stored in no particular order

Dictionary Methods

Experis KickStart ManpowerGroup

- The following table lists some common methods we can use on a dictionary object:

Method	Description
keys()	Returns a sequence containing only the keys from the dictionary
values()	Returns a sequence containing only the values from the dictionary
items()	Returns a sequence of tuples, where each tuple contains a key and value of an element
get(*key*, [*default*])	Returns the value associated with *key.* If *key* is not found it returns None. We can also provide a optional *default* value as the second argument in which case if the key is not found, *default* value will be returned instead of None.
pop(*key*)	Returns the value associated with *key* and removes the specified key and its corresponding value from the dictionary. If *key* doesn't exists KeyError exception is raised.
popitem()	Removes and returns a random element from the dictionary as a tuple
copy()	Creates a new copy of the dictionary
clear()	Removes all elements from the dictionary

get()

- Indexing a dictionary with a key that does not exist is an error
- However, the useful method **get()** can be used to retrieve the value, given a key if it exists, or some default value if it does not
  - If no default is specified, then None is returned
- For example:

```
print(mass.get('Pluto'))
```

```
None
```

```
mass.get('Pluto', -1)
```

```
-1
```

keys, values and items

- The three methods: **keys()**, **values()** and **items()**, return respectively, a dictionary's keys, values and key-value pairs (as tuples)
- In previous versions of Python, each of these methods returned a list, containing a copy of the keys or values, which for most purposes is a wasteful of memory
- In Python 3 these methods return iterable objects
  - This is faster and saves memory

```
planets = mass.keys()
print(planets)
```

```
dict_keys(['Mercury', 'Venus', 'Earth'])
```

```
for planet in planets:
    print(planet, mass[planet])
```

```
Mercury 3.301e+23
Venus 4.867e+24
Earth 5.972e+24
```

keys, values and items

- A dict_keys object can be iterated over any number of times, but it is not a list and cannot be indexed or assigned:

```
planets[0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-57-7c16c1f6b877> in <module>()
----> 1 planets[0]

TypeError: 'dict_keys' object does not support indexing
```

- If you really want a list of the dictionary's keys, simply pass the dict_keys object to the **list()** constructor (which takes any kind of sequence and makes a list out of it):

```
planet_list[1] = 'Jupiter'  # doesn't alter the original dictionary's keys
planet_list
```

```
['Mercury', 'Jupiter', 'Earth']
```

keys, values and items

- Similar methods exist for retrieving a dictionary's values and items (key-value pairs)
  - The objects returned are dict_values and dict_items

```
mass.values()
```

```
dict_values([3.301e+23, 4.867e+24, 5.972e+24])
```

```
mass.items()
```

```
dict_items([('Mercury', 3.301e+23), ('Venus', 4.867e+24), ('Earth', 5.972e+24)])
```

- The items() method returns a sequence of tuples, where each tuple contains a key and a value of an element. We can use a for loop to loop over the tuples as follows:

```
for planet, mass in mass.items():
    print(planet, mass, sep=': ')
```

```
Mercury: 3.301e+23
Venus: 4.867e+24
Earth: 5.972e+24
```

Dictionary Comprehension

- A dictionary comprehension is a construct for creating a dictionary based on another iterable object
- The syntax of dictionary comprehension is:

```
d = {key: value for (key, value) in iterable}
```

  - Each element in the iterable must be iterable itself of two elements
- For example:

```
keys = ['a', 'b', 'c', 'd', 'e']
values = [1, 2, 3, 4, 5]

my_dict = {k: v for (k, v) in zip(keys, values)}
my_dict
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

- It seems require more code than just doing this:

```
dict(zip(keys, values))
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

Dictionary Comprehension

- However, there are more cases when we can utilize the dictionary comprehension
- For instance, we can construct a dictionary from a list using comprehension:

```
{x: x**2 for x in [1, 2, 3, 4, 5]}
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

```
{x.upper(): x * 3 for x in 'abcd'}
```

```
{'A': 'aaa', 'B': 'bbb', 'C': 'ccc', 'D': 'ddd'}
```

Exercise (14)

- Define a function **count_words(text)** that gets a text, and prints the list of words in the text with the number of occurrences of each word
- The words should be printed from the most common word to the least common word
- *Hint*: use Python's string methods to strip out any punctuation
  - It suffices to replace any of the following characters with the empty string: !?":;,()'.*[]
- Example:

```
text = """ "But I don't want to go among mad people," Alice remarked.
"Oh, you can't help that," said the Cat: "we're all mad here. I'm mad. You're mad."
"How do you know I'm mad?" said Alice.
"You must be," said the Cat, "or you wouldn't have come here." """

print(count_words(text))
```

```
mad: 5, you: 4, said: 3, alice: 2, the: 2, cat: 2, here: 2, im: 2, but: 1, i: 1, dont: 1, want: 1, to:
1, go: 1, among: 1, people: 1, remarked: 1, oh: 1, cant: 1, help: 1, that: 1, were: 1, all: 1, youre:
1, how: 1, do: 1, know: 1, must: 1, be: 1, or: 1, wouldnt: 1, have: 1, come: 1, None
```

Exercise (15)

- Anagrams are words that have the same number of same letters, but in different order. For instance:
  - nap - pan
  - ear - are - era
  - cheaters - hectares – teachers
- Write a function **clean_anagrams(words)** that returns a list of words cleaned from anagrams
- For instance:

```
words = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"]
print(clean_anagrams(words))
```

```
['PAN', 'hectares', 'era']
```

Sets

Experis KickStart
ManpowerGroup

- A set is an **unordered collection** of **unique** items
- As with dictionary keys, elements of a set must be hashable objects
- A set is useful for removing duplicates from a sequence and for determining the union, intersection and difference between two collections
- Because they are unordered, set objects cannot be indexed or sliced, but they can be iterated over, tested for membership, and they support the len built-in

Creating Sets

- A set is created by listing its elements between braces {...}

```
s = { 1, 1, 4, 3, 2, 2, 3, 1, "surprise!"}
s
```

```
{1, 2, 3, 4, 'surprise!'}
```

- We can also create sets by passing an iterable to the set() constructor:

```
s2 = set([77, 23, 91, 13])    # create a set from a list
s2
```

```
{13, 23, 77, 91}
```

```
s3 = set("abc123")    # create a set from a string
s3
```

```
{'1', '2', '3', 'a', 'b', 'c'}
```

```
s4 = set(("alpha", "beta", "gamma"))   # create a set from a tuple
s4
```

```
{'alpha', 'beta', 'gamma'}
```

Sets Built-in Functions

- As with list and tuples, we can also use the following functions with sets:

Function	Description
len(*set*)	Returns the number of elements in *set*
sum(*set*)	Returns the sum of elements in the *set*
max(*set*)	Returns the element with the greatest value in *set*
min(*set*)	Returns the element with the smallest value in *set*

```
s1 = {33, 11, 88, 55}
len(s1)
```

```
4
```

```
max(s1)
```

```
88
```

```
min(s1)
```

```
11
```

```
sum(s1)
```

```
187
```

Adding Elements

- The set method **add()** is used to add elements to the set:

```
s = {2, -2, 0}
s.add(1)
s.add(-1)
s.add(1.0)
s
```

```
{-2, -1, 0, 1, 2}
```

  - The last statement doesn't add a new member to the set, since the test 1 == 1.0 is True, so 1.0 is considered to be already in the set

- We can also add multiple elements to the set at once using the **update()** method, which accepts an iterable object:

```
s.update([2, 3, 4])
s
```

```
{-2, -1, 0, 1, 2, 3, 4}
```

Removing Elements

- To remove elements there are several methods:
  - **remove()** removes a specified element but raises a KeyError exception if the element is not present in the set
  - **discard()** does the same, but does not raise an error if the element is not present in the set
  - **pop** (with no argument) removes an arbitrary element from the set
  - **clear()** removes all elements from the set

```
s.remove(1)
s
```

```
{-2, -1, 0, 2, 3, 4}
```

```
s.discard(5)   # OK - does nothing
s
```

```
{-2, -1, 0, 2, 3, 4}
```

```
s.pop()
```

```
0
```

```
s
```

```
{-2, -1, 2, 3, 4}
```

```
s.clear()
s
```

```
set()
```

Looping Through Sets

- Just as with other sequence types, we can use for loop to iterate over the elements of a set:

```
s = {99, 33, 44, 25, 124}
for num in s:
    print(num, end=" ")
```

```
33 99 44 25 124
```

Membership Operators

- As usual, we can use **in** and **not in** operators to find the existence of an element inside a set:

```
s = {99, 33, 44, 25, 124}
33 in s
```

```
True
```

```
5 in s
```

```
False
```

```
5 not in s
```

```
True
```

Set Methods

Experis KickStart ManpowerGroup

- Set objects have a wide range of methods related to properties of mathematical sets:

Method	Description
isdisjoint(*other*)	Is set disjoint with *other*?, i.e., do they have no elements in common?
issubset(*other*) *set* <= *other*	Is *set* a subset of *other*?, i.e., are all the elements of *set* are also elements of *other*?
set < *other*	Is *set* a proper subset of *other*?, i.e., is *set* a subset of *other* but not equal to it?
issuperset(*other*) *set* >= *other*	Is *set* a superset of *other*?
set > *other*	Is *set* a proper superset of *other*?
union(*other*) *set* | *other* | …	The union of *set* and *other*(s), i.e., the set of all elements from both *set* and *other*(s)
intersection(*other*) *set* & *other* & …	The intersection of *set* and *other*(s), i.e., the set of all elements that *set* and *other*(s) have in common
difference(*other*) *set* - *other* - …	The difference of *set* and *other*(s), i.e., the set of elements in *set* but not in *other*(s)
symmetric_difference(*other*) set ^ other ^ …	The symmetric difference of *set* and *other*(s), i.e., the set of elements in either *set* and *other*(s) but not in both

Set Methods

- There are two forms for most set expressions: the operator-like syntax requires all arguments to be set objects, whereas explicit method calls will convert any iterable argument into a set

```
A = {1, 2, 3}
B = {1, 2, 3, 4}
A <= B
```

```
True
```

```
A.issubset([1, 2, 3, 4])  # OK: [1,2,3,4] is turned into set
```

```
True
```

Set Methods

- Some more examples:

```
A = {1, 2, 3}
B = {1, 2, 3, 4}
C = {3, 4, 5, 6}
D = {7, 8, 9}
```

```
B | C   # union
```

```
{1, 2, 3, 4, 5, 6}
```

```
A | C | D   # union of three sets
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
A & C   # intersection
```

```
{3}
```

```
C & D
```

```
set()
```

```
C.isdisjoint(D)
```

```
True
```

```
B - C   # difference
```

```
{1, 2}
```

```
B ^ C   # symmetric difference
```

```
{1, 2, 5, 6}
```

Frozen Sets

- sets are mutable objects, thus they are *unhashable* and so cannot be used as dictionary keys or as members of other sets

```
a = {1, 2, 3}
b = {'q', (1, 2), a}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-39-39b261a243c9> in <module>()
      1 a = {1, 2, 3}
----> 2 b = {'q', (1, 2), a}

TypeError: unhashable type: 'set'
```

- There is, however, a **frozenset** object which is a kind of immutable (and hashable) set
- Frozensets are fixed, unordered collections of unique objects and *can* be used as dictionary keys and set members
  - In a sense, they are to sets what tuples are to lists

Frozen Sets

- For example:

```
a = frozenset((1, 2, 3))
b = {'q', (1, 2), a} #OK: the frozen set is hashable
b
```

```
{(1, 2), frozenset({1, 2, 3}), 'q'}
```

```
b.add(4)  # OK: b is a regular set
```

```
a.add(4)  # Not OK: frozensets are immutable
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-45-209e4590c6b4> in <module>()
----> 1 a.add(4)  # Not OK: frozensets are immutable

AttributeError: 'frozenset' object has no attribute 'add'
```

Exercise (16)

- Write a function, using set objects, to remove duplicates from a list
- For example:

```
remove_duplicates([5,7,4,4,5,4,2,8,3,8])
```

```
[2, 3, 4, 5, 7, 8]
```

Exercise (17)

Experis KickStart ManpowerGroup

- A *Mersenne prime*, M_i, is a prime number of the form $M_i = 2^i - 1$
- Mersenne primes are noteworthy due to their connection to perfect numbers
  - All even perfect numbers have the form $2^{p-1}(2^p - 1)$ where $2^p - 1$ is prime
- The set of Mersenne primes less than or equal to *n* may be thought of as the intersection of the set of all primes less than or equal to *n*, P_n, with the set, A_n, of integers satisfying $2^i - 1 \leq n$
- Write a program that prints the list of the Mersenne primes less than 1,000,000
- Hint: Use the Sieve of Eratosthenes method to find all primes less than or equal to *n*,

Exercise (18)

- The *power set* of a set *S*, *P*(*S*), is the set of all subsets of *S*, including the empty set and *S* itself
- For example, *P*({1,3,6}) = {{}, {1}, {3}, {6}, {1,3}, {1,6}, {3,6}, {1,3,6}}
- Write a generator function that returns the power set of a given set
- Usage example:

```
my_set = set([1, 3, 6])
```

```
for s in power_set(my_set):
    print(s)
```

```
set()
{1}
{3}
{1, 3}
{6}
{1, 6}
{3, 6}
{1, 3, 6}
```

File Handling

Experis KickStart ManpowerGroup

- Until now, data has been hard-coded into our Python programs, and output has been to the console (or the notebook)
- Of course, it will frequently be necessary to input data from an external file and to write data to an output file
- In Python, file handling consists of three steps:
  - Opening the file
  - Processing the file, i.e., performing read or write operations
  - Closing the file

File Types

- There are two major types of files:
  - **Text files** – files whose contents can be viewed using a text editor
    - A text file is a sequence of ASCII or Unicode characters
    - Examples for text files include Python programs, HTML pages, and text documents
  - **Binary files** – files that store the data in the same way as it is stored in memory
    - You can't read a binary file using a text editor
    - Examples for binary files include mp3 files, image files and word documents
- Storing the number 1234 in a binary vs a text file:

	byte 0	byte 1	byte 2	byte 3
Binary:	00000000	00000000	00000100	11010010
	0	0	4	210
Text:	49	50	51	52

Opening a File

- The function **open()** is used to open a file and create a file object
- Its syntax is: `fileobject = open(filename, mode)`
  - *filename* may be given as an absolute path, or as a path relative to the directory in which the program is being executed
  - *mode* is a string that specifies the type of the operation you want to perform on the file
- For example, to open a file for text-mode writing:

```
f = open("myfile.txt", "w")
```

- In Windows, remember to escape backslashes while using absolute path names:

```
f = open("C:\\Users\\roi\\Documents\\README.md", "w")
```

- Or you can use a “raw string” by specifying the **r** character in front of the string:

```
f = open(r"C:\Users\roi\Documents\README.md", "w")
```

File Modes

Experis KickStart ManpowerGroup

- The following table lists the different modes available to you:

mode argument	Description
r	text, read-only (the default)
w	text, write (an existing file with the same name will be overwritten)
a	text, append to an existing file
r+	text, reading and writing
rb	binary, read-only
wb	binary, write (an existing file with the same name will be overwritten)
ab	binary, append to an existing file
rb+	binary, reading and writing

Closing a File

- File objects are closed with the **close()** method as follows:

```
f.close()
```

- Closing a file releases valuable system resources
- Python closes any open file objects automatically when a program terminates or when the file object is no longer referenced in the program
- However, it is a good practice to close a file once you are done working with it

Writing to a Text File

- The **write()** method of a file object writes a *string* to the file and returns the number of characters written:

```
f = open("myfile.txt", "w")
```

```
f.write("Hello world")
```

```
11
```

```
f.flush()
```

- The characters are written to an internal buffer before they are written to the file
- The method **flush()** flushes the internal buffer
  - Python automatically flushes the files when closing them
- You can check the default buffer size by calling:

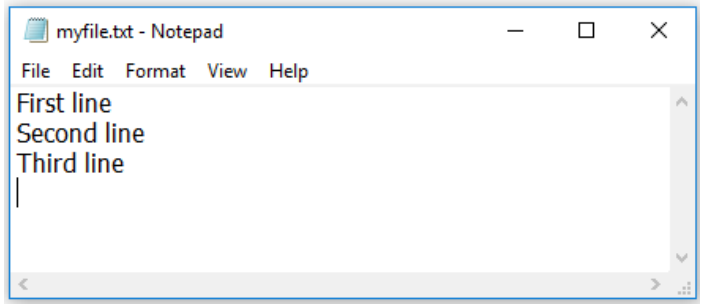
```
import io
print (io.DEFAULT_BUFFER_SIZE)
```

```
8192
```

Writing to a Text File

- Note that unlike print() function, the write() method doesn't print a newline character (\n) at the end of the string
- If you want to write multiple lines to the file, you should append \n to each line:

```
f = open("myfile.txt", "w")

f.write("First line\n")
f.write("Second line\n")
f.write("Third line\n")

f.close()
```



Writing to a Text File

Experis KickStart ManpowerGroup

- More helpfully, the **print()** built-in takes an argument, *file*, to specify where to redirect its output:

```
f = open("myfile.txt", "w")

print("First line", file=f)
print("Second line", file=f)
print("Third line", file=f)

f.close()
```

- This program produces the same output as before, the only difference is that here the newline character (\n) is automatically added by the print() function.

Reading from a Text File

Experis KickStart ManpowerGroup

- There are a few methods available for reading from a file:
  - **read(n)** – reads n bytes from the file
    - In ASCII text files, this means reading n characters from the file
    - If n is omitted, the entire file is read in
  - **readline()** – reads a single line the file, up to and including the newline character
  - **readlines()** – reads all of the lines into a list of strings in one go
- For example, to read all the file at once:

```
f = open("myfile.txt", "r")

print(f.read())   # read all content at once

f.close()
```

```
First line
Second line
Third line
```

Reading from a Text File

Experis KickStart ManpowerGroup

- Example for reading the data in chunks:

```
f = open("myfile.txt", "r")

print("First chunk: ", f.read(4))    # read the first 4 characters
print("Second chunk: ", f.read(10))  # read the next 10 characters
print("Third chunk: ", f.read())     # read the reamining characters

f.close()
```

```
First chunk:  Firs
Second chunk:  t line
Sec
Third chunk:  ond line
Third line
```

Reading from a Text File

- Example for reading the file line by line using **readline()**:

```
f = open("myfile.txt", "r")

line = f.readline()
while line:
    print(line, end="")
    line = f.readline()

f.close()
```

```
First line
Second line
Third line
```

  - Both read() and readline() return an empty string when they reach the end of the file
  - Because line retains its newline character when read in, we use end="" to prevent print from adding another, which would be output as a blank line

Reading from a Text File

- File objects are iterable, and looping over a (text) file returns its lines one at a time:

```
f = open("myfile.txt", "r")

for line in f:
    print(line, end="")

f.close()
```

```
First line
Second line
Third line
```

Appending Data to a Text File

- We can use "a" mode to append data to end of the file
- The following program demonstrates how to append data to the end of the file:

```
f = open("myfile.txt", "a")

print("Fourth line", file=f)
print("Fifth line", file=f)

f.close()
```

```
f = open("myfile.txt", "r")

for line in f:
    print(line, end="")

f.close()
```

```
First line
Second line
Third line
Fourth line
Fifth line
```

Exercise (19)

- Write a program that writes the first four powers of the numbers between 1 and 1,000 in comma-separated fields to the file powers.txt
- The file contents should look like:

```
1, 1, 1, 1
2, 4, 8, 16
3, 9, 27, 81
...
999, 998001, 997002999, 996005996001
1000, 1000000, 1000000000, 1000000000000
```

- Then read the numbers from the file powers.txt and print a list of the cubes of all the numbers between 1 and 100

Exercise (20)

- The coast redwood tree species, *Sequoia sempervirens*, includes some of the oldest and tallest living organisms on Earth
- Some details concerning individual trees are given in the tab-delimited text file redwood-data.txt at https://goo.gl/KnFPfC
- Write a Python program to read in this data and report the tallest tree and the tree with the greatest diameter

Exercise (21)

- The novel *Moby-Dick* is out of copyright and can be downloaded as a text file from the Project Gutenberg website at http://www.gutenberg.org/files/2701/
- Write a program to output the 100 words most frequently used in the book by storing a count of each word encountered in a dictionary
- *Hint*: use Python's string methods to strip out any punctuation
  - It suffices to replace any of the following characters with the empty string: !?":;,()'.*[]
- Compare the frequencies of the top 100 words with the prediction of *Zipf's Law*:

$$\log f(w) = \log C - a \log r(w)$$

  - where $f(w)$ is the number of occurrences of word w, $r(w)$ is the corresponding rank (1 = most common, 2 = second most common, etc.) and C and a are constants
  - Typically, $C = \log f(w_1)$, where w_1 is the most common word, and $a = 1$

The with Statement

Experis KickStart ManpowerGroup

- The **with** statement creates a block of code that is executed within a certain *context*
- A context is defined by a *context manager* that provides a pair of methods (__enter__ and __exit__) describing how to enter and leave the context
- For example, the file object returned by the open() method is a context manager
- It defines an exit method which is called on exiting the context, and simply closes the file, so this doesn't need to be done explicitly
- To open a file within a context, use:

```
with open(filename, mode) as f:
    # process the file in some way, for example:
    lines = f.readlines()
```

  - The scope of the file object f is limited to the body of the with statement
  - The context manager ensures that the file is closed, even if something goes wrong in the block

The with Statement

- The following example shows how to read a file within a with statement:

```
with open("myfile.txt", "r") as f:
    for line in f:
        print(line, end="")
```

```
First line
Second line
Third line
Fourth line
Fifth line
```

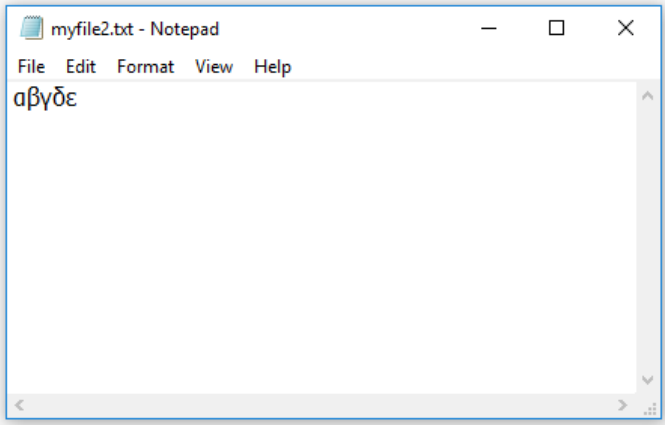
Working with Unicode Text Files

- To work with Unicode text files, you need to open the file using the open() function from the **io** module, and specify the *encoding* argument:

```
import io

with io.open("myfile2.txt", "w", encoding="utf8") as f:
    print("αβγδε", file=f)
```

```
with io.open("myfile2.txt", "r", encoding="utf8") as f:
    print(f.read())
```

```
αβγδε
```



Working with Binary Files

Experis KickStart ManpowerGroup

- To write to a binary file, you need to open the file for writing in binary mode and then write data to the file as hexadecimal strings:

```
f = open("myfile.bin", "wb")

f.write(b"\x0a\x1b\x2c")
f.write(b"\x3d\x4e\x5f")

f.close()
```

- To convert Python values to hexadecimal strings, you need to use the **struct** module
- The struct module performs conversions between Python values and C structs represented as Python strings
  - This module is useful in handling binary data stored in files or from network connections among other sources

Working with Binary Files

- **struct.pack**(*fmt*, *v1*, *v2*, ...) returns a string containing the values v1, v2, ... packed according to the given format
- **struct.unpack**(*fmt*, *string*) unpacks the string according to the given format
  - The result is a tuple even if it contains exactly one item
  - The string must contain exactly the amount of data required by the format

```
import struct

f = open("myfile.bin", "wb")
f.write(struct.pack('I', 23250))
f.write(struct.pack('?', True))
f.write(struct.pack('5s', "Hello".encode()))

f.close()
```

```
f = open("myfile.bin", "rb")

print(struct.unpack('I', f.read(4))[0])
print(struct.unpack('?', f.read(1))[0])
print(struct.unpack('5s', f.read(5))[0].decode("UTF-8"))

f.close()
```

```
23250
True
Hello
```

Format Characters

- The format strings describe the layout of the C structs and the intended conversion to/from Python values:

Format	C Type	Python type	Standard size
x	pad byte	no value	
c	char	string of length 1	1
b	signed char	integer	1
B	unsigned char	integer	1
?	_Bool	bool	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4
I	unsigned int	integer	4
l	long	integer	4
L	unsigned long	integer	4
q	long long	integer	8
Q	unsigned long long	integer	8
f	float	float	4
d	double	float	8
s	char[]	string	
p	char[]	string	
P	void *	integer	

Working with Binary Files

Experis KickStart ManpowerGroup

- To write to a binary file, you need to open the file for writing in binary mode and then write data to the file as hexadecimal strings:

```
f = open("myfile.bin", "wb")

f.write(b"\x0a\x1b\x2c")
f.write(b"\x3d\x4e\x5f")

f.close()
```

- To convert Python values to hexadecimal strings, you need to use the **struct** module
- The struct module performs conversions between Python values and C structs represented as Python strings
  - This module is useful in handling binary data stored in files or from network connections among other sources

Exercise (22)

- Define a function **copy_file(source, dest)** that takes the paths of a source file and a destination file, and copies the binary data from the source file to the destination file
- Note: the source file could be large, and might not fit entirely in memory
- Usage example:

```
copy_file("source.jpg", "dest.jpg")
```

```
2048 bytes copied successfully
2048 bytes copied successfully
2048 bytes copied successfully
2048 bytes copied successfully
2048 bytes copied successfully
2048 bytes copied successfully
700 bytes copied successfully
```

Errors and Exceptions

Experis KickStart ManpowerGroup

- Python distinguishes between two types of error:
- **Syntax errors** are mistakes in the grammar of the language and are checked before the program is executed
  - Syntax errors are always fatal: there is nothing the Python compiler can do for you if your program does not conform to the grammar of the language
- **Exceptions** are *runtime errors*: conditions usually caused by attempting an invalid operation on an item of data (such as division by zero)
  - Mechanisms exist for "catching" runtime errors and the condition gracefully without stopping the program's execution

Syntax Errors

- Syntax errors are caught by the Python compiler and produce a message indicating where the error occurred
- For example:

```
for lambda in range(8):
```

```
  File "<ipython-input-1-f6fc4897d4ad>", line 1
    for lambda in range(8):
               ^
SyntaxError: invalid syntax
```

  - Because lambda is a reserved keyword, it cannot be used as a variable name. Its occurrence where a variable name is expected is therefore a syntax error.

Syntax Errors

- Because a line of Python code may be split within an open bracket (“()”, “[]”, or “{}”), a statement split over several lines can sometimes cause a SyntaxError to be indicated somewhere other than the location of the true bug:

```
a = [1, 2, 3, 4,
b = 5
```

```
  File "<ipython-input-2-2ff941d04ea6>", line 2
    b = 5
      ^
SyntaxError: invalid syntax
```

  - Here, the statement b = 5 is syntactically valid: the error arises from failing to close the square bracket of the previous list declaration
- There are two special types of SyntaxError that are worth mentioning:
  - **IndentationError** occurs when a block of code is improperly indented
  - **TabError** is raised when a tabs and spaces are mixed inconsistently to provide indentation

Exceptions

Experis KickStart ManpowerGroup

- An exception occurs when an syntactically correct expression is executed and causes a **runtime error**
- There are different types of built-in exceptions, and custom exceptions can be defined by the programmer if required
- If an exception is not “caught” using the try ... except clause described later, it halts the execution of the program
- When a runtime error occurs, Python creates an exception object which contains all the relevant information about the error just occurred
- This object contains an error message and a **stack traceback**: the history of function calls leading to the error is reported so that its location in the program execution can be determined

Common Python Exceptions

Experis KickStart ManpowerGroup

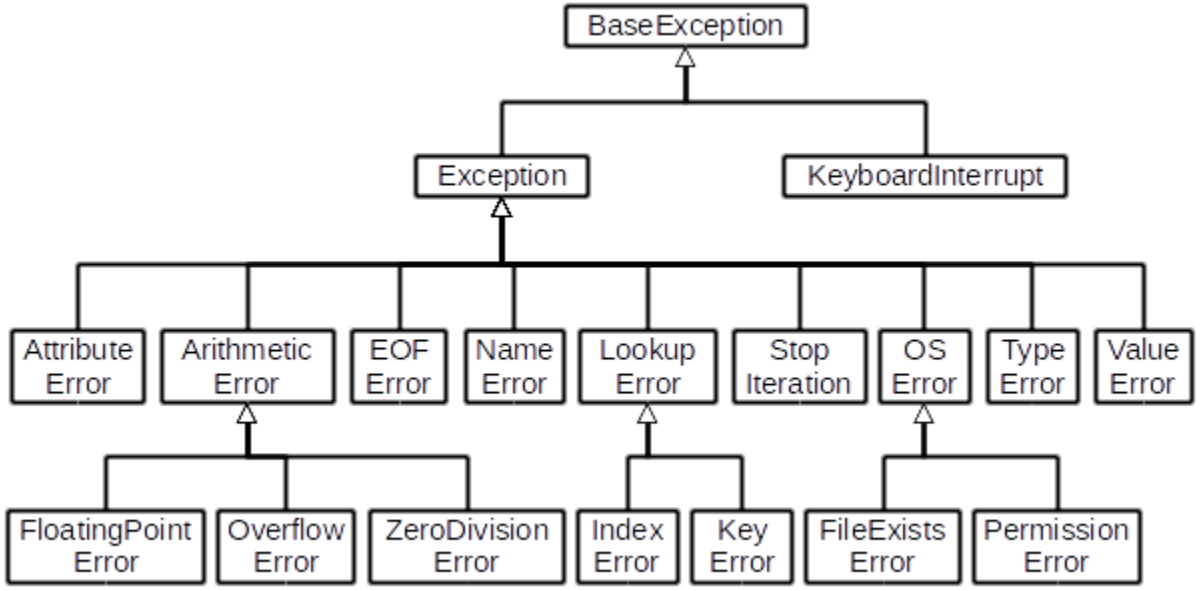
- A list of the more commonly encountered built-in exceptions and their descriptions:

Exception	Cause and Description
FileNotFoundError	Attempting to open a file or directory that does not exist, inherits from OSError
IndexError	Indexing a sequence (such as a list or string) with a subscript that is out of range
KeyError	Indexing a dictionary with a key that does not exist in that dictionary
NameError	Referencing a local or global variable name that has not been defined
TypeError	Attempting to use an object of an inappropriate type as an argument to a built-in operation or function
ValueError	Attempting to use an object of the correct type but with an incompatible value as an argument to a built-in operation or function
ZeroDivisionError	Attempting to divide by zero
SystemExit	Raised by the sys.exit function – if not handled, this function causes the Python interpreter to exit

- The full list can be found at https://docs.python.org/3/library/exceptions.html

Exception Class Hierarchy

- The **BaseException** class is the root of all exception classes in Python
- The following figure shows exception class hierarchy in Python:



Common Python Exceptions

Experis KickStart ManpowerGroup

- A **NameError** exception occurs when a variable name is used that hasn't been:

```
print("x = ", x)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-189b4fb9adac> in <module>()
----> 1 print("x = ", x)

NameError: name 'x' is not defined
```

- A division by zero causes a **ZeroDivisionError**:

```
a, b = 5, 0
a / b
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-a827eb207e9c> in <module>()
      1 a, b = 5, 0
----> 2 a / b

ZeroDivisionError: division by zero
```

Common Python Exceptions

Experis KickStart ManpowerGroup

- Trying to access an element at invalid index causes an **IndexError**:

```
list1 = [11, 3, 99, 15]
list1[5]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-3-16d803b33219> in <module>()
      1 list1 = [11, 3, 99, 15]
----> 2 list1[5]

IndexError: list index out of range
```

- Trying to open a file that doesn't exist in read mode causes a **FileNotFoundError**:

```
f = open("filedoesnexist.txt", "r")
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-4-00ca043a59e9> in <module>()
----> 1 f = open("filedoesnexist.txt", "r")

FileNotFoundError: [Errno 2] No such file or directory: 'filedoesnexist.txt'
```

Common Python Exceptions

Experis KickStart ManpowerGroup

- A **TypeError** is raised if an object of the wrong type is used in an expression or function, for example when trying to add a string to an integer:

```
10 + "12"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-db2f06a66b01> in <module>()
----> 1 10 + "12"

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- A **ValueError**, on the other hand, occurs when the object involved has the correct *type* but an invalid *value*:

```
float("hello")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-7-7124e8e12e61> in <module>()
----> 1 float("hello")

ValueError: could not convert string to float: 'hello'
```

Traceback Report

- When an exception is raised but not handled, Python will issue a traceback report indicating where in the program flow it occurred:

```
def lumberjack():
    bright_side_of_death()

def bright_side_of_death():
    return tuple()[0]

lumberjack()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-10-81211c2c9fdb> in <module>()
      5     return tuple()[0]
      6
----> 7 lumberjack()

<ipython-input-10-81211c2c9fdb> in lumberjack()
      1 def lumberjack():
----> 2     bright_side_of_death()
      3
      4 def bright_side_of_death():
      5     return tuple()[0]

<ipython-input-10-81211c2c9fdb> in bright_side_of_death()
      3
      4 def bright_side_of_death():
----> 5     return tuple()[0]
      6
      7 lumberjack()

IndexError: tuple index out of range
```

Handling Exceptions

Experis KickStart ManpowerGroup

- To catch an exception in a block of code, write the code within a **try:** clause and handle any exceptions raised in an **except:** clause, using the following syntax:

```
try:
    # try block
    # write code that might raise an exception here
    <statements>
except ExceptionType:
    # except block
    # handle exception here
    <exception handler>
```

  - When an exception occurs in the try block, execution of the rest of the try block is skipped
    - If the exception raised matches the exception type in the except clause, the corresponding handler is executed
    - If the exception doesn't match the exception type in the except clause, the program halts with a traceback report
  - On the other hand, If no exception is raised in the try block, the except clause is skipped

Handling Exceptions

- The following example catches a ZeroDivisionError exception:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result: ", result)
except ZeroDivisionError:
    print("Can't divide a number by zero")
```

```
Enter a number: 0
Can't divide a number by zero
```

- If we run the program again and enter a string instead of a number, then a ValueError exception is raised, which is *unhandled* by our except clause:

```
Enter a number: hello

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-3-834c05ed2c52> in <module>()
      1 try:
----> 2     num = int(input("Enter a number: "))
      3     result = 10 / num
      4     print("Result: ", result)
      5

ValueError: invalid literal for int() with base 10: 'hello'
```

Handling Multiple Exceptions

Experis KickStart ManpowerGroup

- We can add as many except clauses as we want to handle different types of exceptions. The general format of such a try-except statement is as follows:

```
try:
    # write code that might raise an exception here
except <ExceptionType1>:
    # handle ExceptionType1 here
except <ExceptionType2>:
    # handle ExceptionType2 here
…
except:
    # handle any other type of exception here
```

  - When exception occurs, Python matches the exception raised against every except clause sequentially
  - If a match is found then the handler in the corresponding except clause is executed and rest of the of the except clauses are skipped
  - In case the exception raised doesn't match any except before the last except clause, then the handler in the last except clause is executed

Handling Multiple Exceptions

Experis KickStart ManpowerGroup

- Example for a try block with multiple except blocks:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result: ", result)
except ZeroDivisionError:
    print("Can't divide a number by zero")
except ValueError:
    print("Only integers are allowed")
except:
    print("Some unexpected error occurred")
```

```
Enter a number: hello
Only integers are allowed
```

Handling Multiple Exceptions

- Another example for handling exceptions when reading data from a file:

```
filename = input("Enter file name: ")

try:
    f = open(filename, "r")
    for line in f:
        print(line, end="")
    f.close()
except FileNotFoundError:
    print("File not found")
except PermissionError:
    print("You don't have permission to read the file")
except:
    print("Unexpected error while reading the file")
```

```
Enter file name: test
File not found
```

Handling Multiple Exceptions

- To handle more than one exception in a single except block, list them in a tuple (which must be within brackets)

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result: ", result)

except (ZeroDivisionError, ValueError):
    print("The input is zero or not numeric!")
```

```
Enter a number: hello
The input is zero or not numeric!
```

The else and finally clauses

Experis KickStart ManpowerGroup

- The try ... except statement has two more optional clauses (which must follow any except clauses):
  - Statements in a block following the **else** keyword are executed if an exception was not raised
  - Statements in a block following the **finally** keyword are always executed, whether an exception was raised or not
- The use of the **else** clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try ... except statement
- The finally block is used to make sure files or resources are closed or released regardless of whether an exception occurs, even if you don't catch the exception

The else and finally clauses

Experis KickStart ManpowerGroup

- The following code gives an example of a try ... except ... else ... finally clause:

```
def process_file(filename):
    try:
        f = None
        f = open(filename, "r")
    except IOError:
        print("Couldn't open {} for reading".format(filename))
        return
    else:
        # we don't want to catch IOError here if it's raised
        for line in f:
            print(line, end="")
    finally:
        # close the file if it was successfully opened
        if f:
            f.close()
```

```
process_file("test.txt")
```

```
Couldn't open test.txt for reading
```

```
process_file("myfile.txt")
```

```
First line
Second line
Third line
```

Ask Forgiveness, Not Permission

Experis KickStart ManpowerGroup

- **EAFP** (easier to ask for forgiveness than permission) is a common Python coding style that assumes the existence of files, attributes, valid keys, etc., and catches exceptions if the assumption proves false
- This clean style is characterized by the presence of many try and except statements:

Ask for permission

```
if can_do_operation():
    perform_operation()
else:
    handle_error_case()
```

Ask for forgiveness

```
try:
    perform_operation()
except UnableToPerform:
    handle_error_case()
```

- E.g., asking for forgiveness when accessing dictionary keys that may not exist:

```
contacts = {"Tom": "122-444-333", "Jim": "412-1231-121", "Ron": "891-121-1212"}

name = input("Enter a name: ")
try:
    print(contacts[name])
except KeyError:
    print("Sorry, there is no contact with the name {}".format(name))
```

```
Enter a name: Joe
Sorry, there is no contact with the name Joe
```

Don’t Do

Experis KickStart ManpowerGroup

- You may come across the following type of construction:

```
try:
    [do something]
except:                # Don't do this!
    pass
```

- This will execute the statements in the try block and ignore *any* exceptions raised – it is very unwise to do this as it makes code very hard to maintain and debug
- Always catch specific exceptions and handle them appropriately, allowing any other exceptions to “bubble up” to be handled (or not) by any other except clauses

Exercise (25)

Experis KickStart ManpowerGroup

- Write a program to read in the data from the file swallow-speeds.txt at https://goo.gl/KnFPfC and use it to calculate the average air-speed velocity of an (unladen) African swallow
- Use exceptions to handle the processing of lines that do not contain valid data points

Raising Exceptions

Experis KickStart ManpowerGroup

- Sometimes it is desirable for a program to raise a particular exception if some condition is met
- A function raises exception by creating an exception object from the appropriate class and using the **raise** keyword to throw the exception to the calling code:

```
raise SomeExceptionClass("Error message describing cause of the error")
```

- When an exception is raised inside a function and is not caught there, it is automatically propagated to the calling function (and any function up in the stack), until it is caught by a try-except statement in some calling function
- If the exception reaches the main module and still not handled, the program terminates with an error message

Raising Exceptions

Experis KickStart ManpowerGroup

- For example, let's define a function to calculate the factorial of a number
- Factorial is only valid for positive integers, thus we can prevent passing data of any other type by checking the argument and raising the appropriate exception:

```
def factorial(n):
    if type(n) is not int:
        raise TypeError("Argument must be int")

    if n < 0:
        raise ValueError("Argument must be non-negative")

    f = 1
    for i in range(2, n + 1):
        f *= i
    return f
```

```
try:
    print("Factorial of 4 is:", factorial(4))
    print("Factorial of 1.5 is:", factorial(1.5))
except Exception:
    print("Invalid input")
```

```
Factorial of 4 is: 24
Invalid input
```

Accessing the Exception Object

- We can access the exception object using the following form of except clause:

```
except ExceptionType as e
```

  - The exception object will be assigned to the variable e
- For example:

```
try:
    print("Factorial of 4 is:", factorial(4))
    print("Factorial of 1.5 is:", factorial(1.5))
except Exception as e:
    print("Error:", e)
```

```
Factorial of 4 is: 24
Error: Argument must be int
```

  - Notice that the error message printed by the exception object is the same message that we passed while raising the exception

Exercise (26)

Experis KickStart ManpowerGroup

- Python follows the convention of many computer languages in choosing to define $0^0 = 1$
- Write a function, **power(a, b),** which behaves the same as the Python expression a**b, but raises a ValueError if a and b are both zero
- Also if a or b are not numbers, the function should raise a TypeError
- Write a proper docstring for the function including the description of the raised exceptions

Modules

Experis KickStart ManpowerGroup

- As we've seen, Python is quite a modular language and has functionality beyond the core programming essentials
- In programming, a **module** is a piece of software that has a specific functionality
- Modules are a good way to organize large programs in a hierarchical way
- For example, when building a game, one module would be responsible for the game logic, another module would be responsible for drawing the game on the screen, etc.
- Each module is written in a different file, which can be edited separately

Python Modules

Experis KickStart ManpowerGroup

- A module in Python is simply a Python file that ends with .py extension
- The name of the module will be the name of the file
  - For example, a Python file called hello.py has the module name of hello that can be imported into other Python files or used on the Python command line interpreter
- Modules can define functions, classes, and variables that you can reference in other Python files
- In the following example we will create two modules in the folder mygame:

```
mygame/
      game.py
      ui.py
```

  - The Python script game.py implements the game logic
  - It will use functions from ui.py, which is responsible for drawing the game on screen and interacting with the user

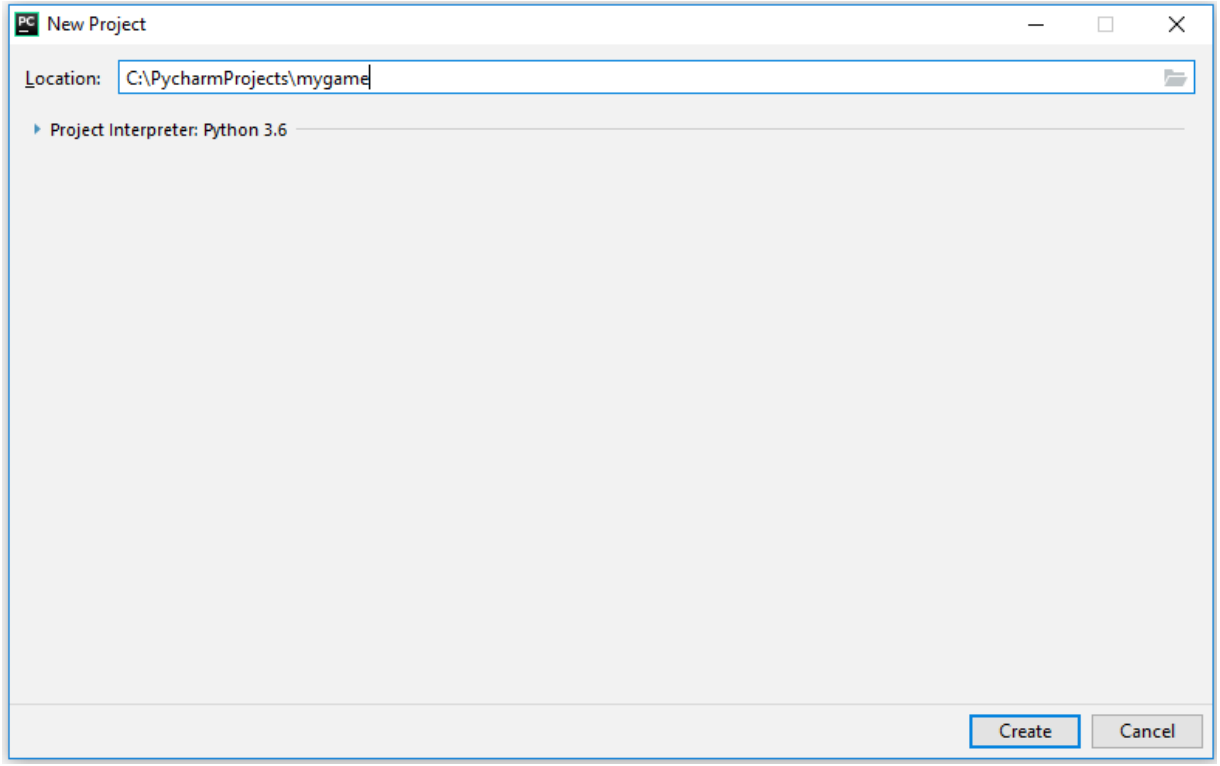
Module Names

Experis KickStart ManpowerGroup

- Note that because of the syntax of the import statement, you should avoid naming your module anything that isn't a valid Python identifier
- For example, the filename <module>.py should not contain a hyphen or start with a digit
- Do not give your module the same name as any built-in modules (such as math or random) because these get priority when Python imports

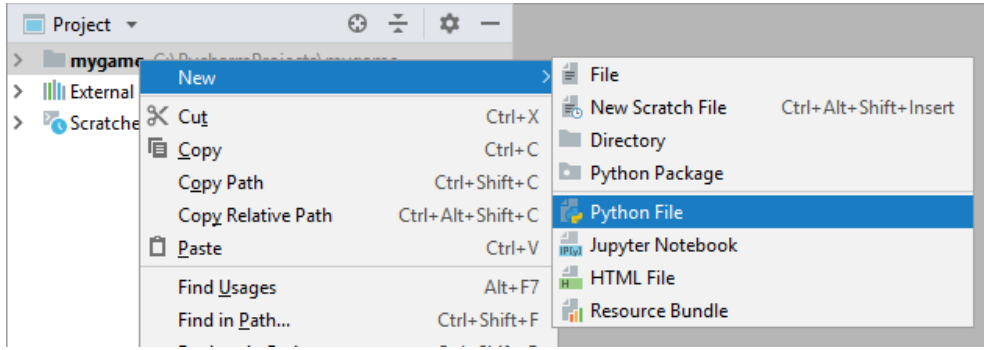
Create a New PyCharm Project

- Create a new project called mygame in PyCharm:

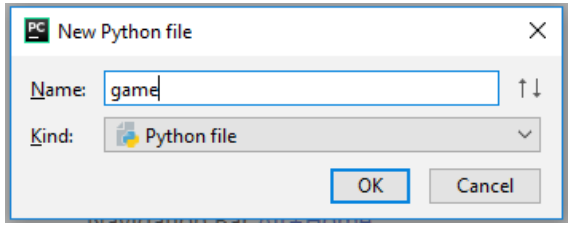


Creating a Module

- Add a new Python script to the project, by right-clicking the project name in the Project pane and choosing New -> Python File



- Name the file game.py



- Similarly, add another module named ui.py

Creating a Module

- Enter the following code in ui.py:

```
# ui.py

def print_welcome_message():
    print("Welcome to my amazing game!")
    print("Press ENTER to start")
    input()

def draw_board():
    print("======================")
    print("======================")
    print("===== Game board =====")
    print("======================")
    print("======================")
    print()

def get_move(player_num):
    print("Player {}: it's your turn".format(player_num))
    move = input("Enter your move: ")
    return move
```

Importing a Module

- To use functions, constants or classes defined inside a module, we first have to **import** it using the import statement
- The syntax of the import statement is:

```
import module_name
```

- Upon encountering the line import <module>, the Python interpreter:
  - Executes the statements in the file <module>.py
  - Enters the module name <module> into the current namespace, so that the attributes it defines are available with the “dotted syntax”: <module>.<attribute>

Importing a Module

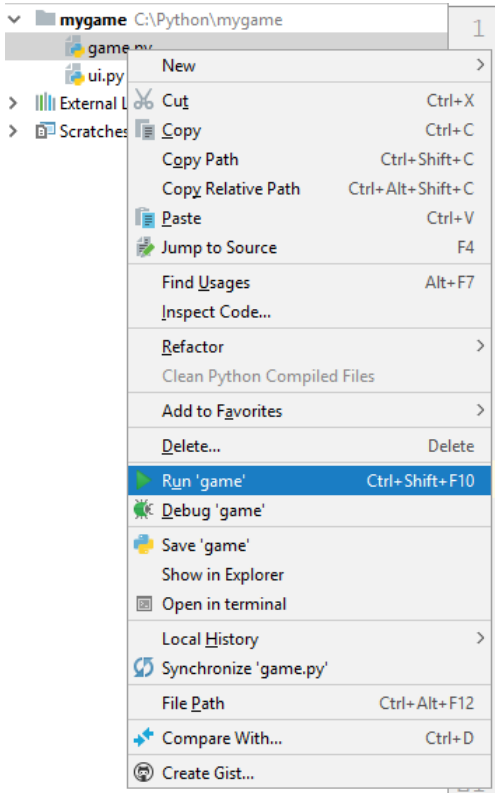
- Since game.py uses the functions defined in ui.py, it should first import it:

```
# game.py

# import the ui module
import ui

def start():
    ui.print_welcome_message()
    play_game()

def play_game():
    current_player = 0
    move = None
    while move != "exit":
        ui.draw_board()
        move = ui.get_move(current_player)
        current_player = 1 - current_player

if __name__ == "__main__":
    start()
```

This means that main() will be executed only when the module is run directly, but not when it is imported by another module

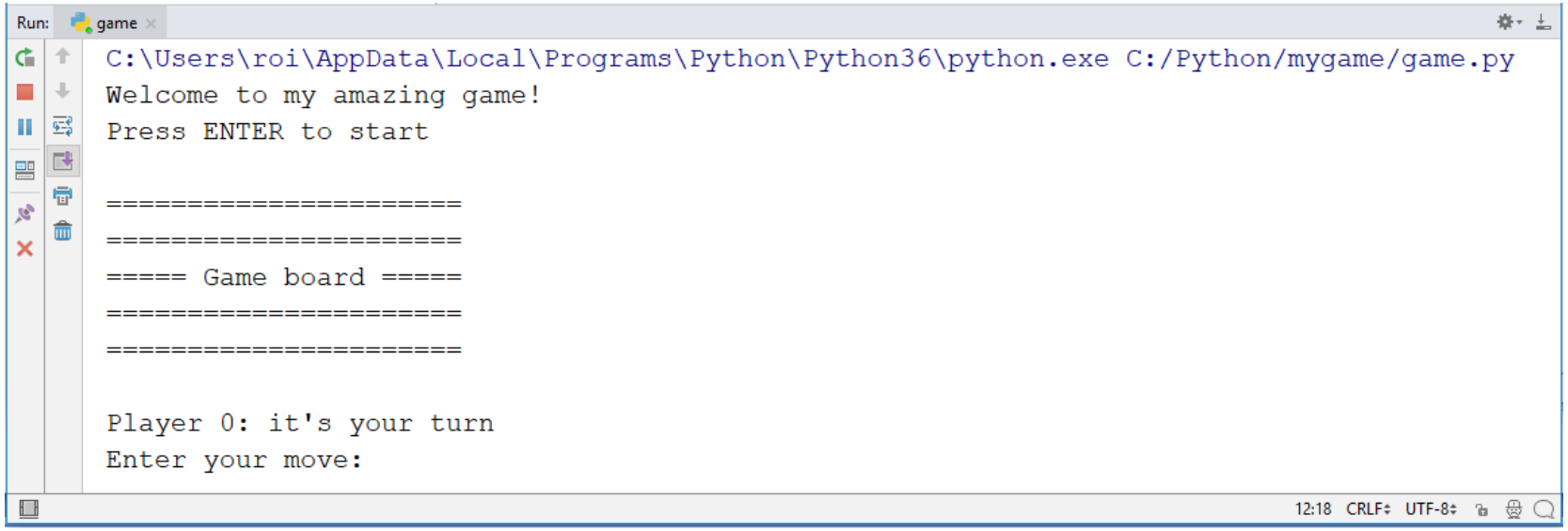
Running a Module

- You can run a module by right-clicking its name in the Project pane and choosing Run… (or pressing Ctrl+Shift+F10)



Running a Module

- A Run pane will open where you can see the program output:



Importing Selected Identifiers

Experis KickStart ManpowerGroup

- The statement **import my_module** imports every identifier in the module
- To use only specific names from the module, use the following import statement:

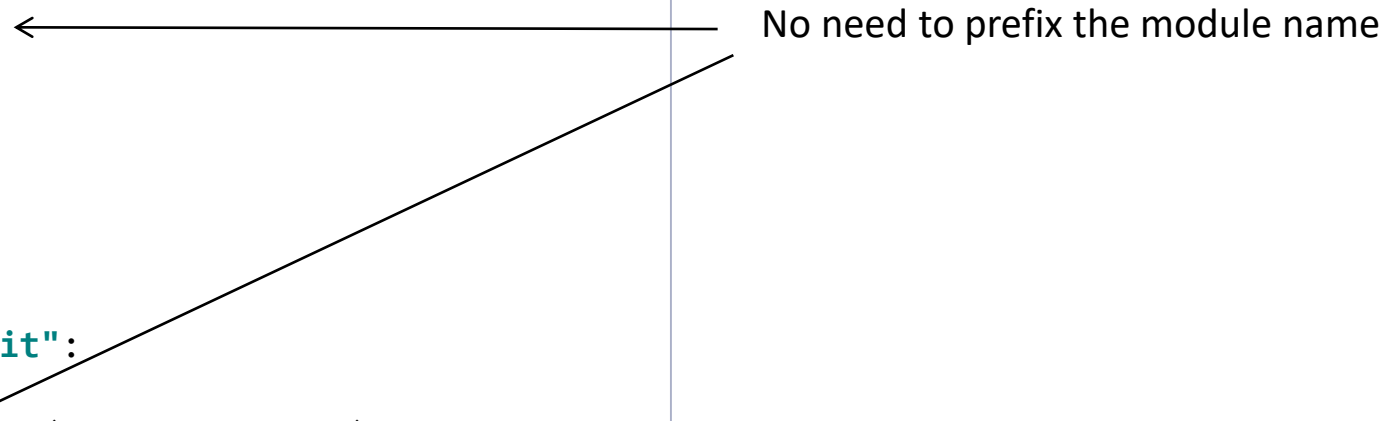
```
from module_name import name1, name2, ..., nameN
```

- Any code after this import statement can use name1, name2, etc. without prefixing them with the module name

Importing Selected Identifiers

- For example, game.py could import the specific functions it needs from ui.py:

```
# game.py

from ui import print_welcome_message, draw_board, get_move

def start():
    print_welcome_message()
    play_game()

def play_game():
    current_player = 0
    current_move = None

    while current_move != "exit":
        draw_board()
        current_move = get_move(current_player)
        current_player = 1 - current_player

if __name__ == "__main__":
    start()
```



Importing All Identifiers

Experis KickStart ManpowerGroup

- You may also import every identifier from a module by using the import * command:

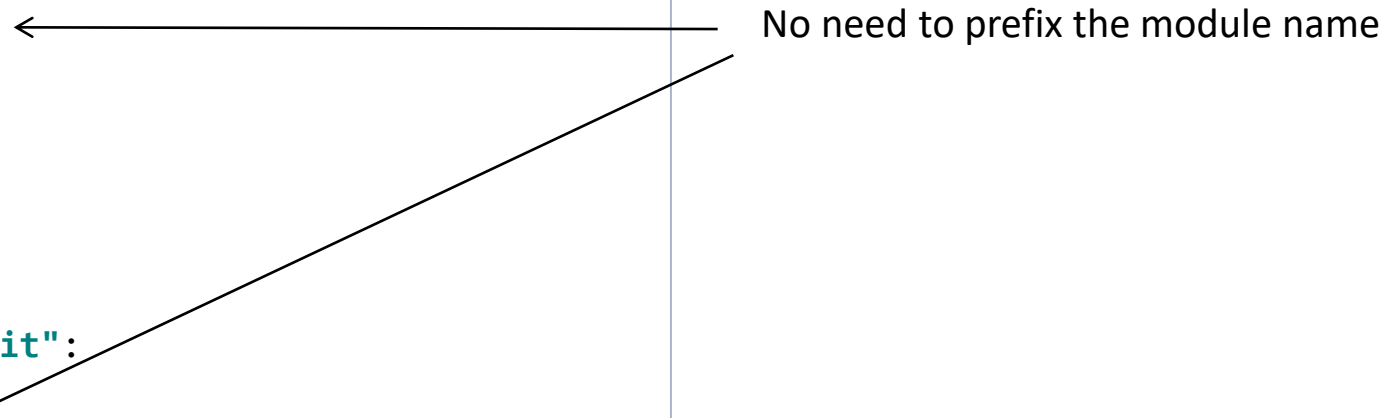
```
from module_name import *
```

- This statement is functionally equivalent to import module_name
- However, import * allows to access identifiers in the module without prefixing them with the module name
- By default, import * imports all identifiers in the module not starting with an underscore
  - You can override this behavior by assigning the variable **__all__** to a list of identifiers that shall be loaded and imported when import * is used
- In general using import * is not recommended, since there is a danger of name conflicts (particularly if many modules are imported in this way)

Importing All Identifiers

- For example, game.py could use import * to import all identifiers from ui.py:

```
# game.py

from ui import *

def start():
    print_welcome_message()
    play_game()

def play_game():
    current_player = 0
    current_move = None

    while current_move != "exit":
        draw_board()
        current_move = get_move(current_player)
        current_player = 1 - current_player

if __name__ == "__main__":
    start()
```



Aliasing Modules

- It is possible to modify the names of modules and their functions within Python by using the **as** keyword
- You may want to change a name because:
  - You've already used the same name for something else in your program
  - Another module you have imported also uses that name
  - You may want to abbreviate a longer name that you are using a lot
- The construction of this statement looks like this:

```
import module_name as another_name
```

- For example, we could abbreviate the name of the math module to m:

```
import math as m
```

```
print(m.pi)
print(m.e)
```

```
3.141592653589793
2.718281828459045
```

Aliasing Modules

- You can also define an alias to a selected identifier imported from the module
- For example:

```
from math import radians as rad
```

```
print(rad(90))
```

```
1.5707963267948966
```

- For some modules, it is commonplace to use aliases
- For example, the matplotlib.pyplot module's official documentation calls for use of plt as an alias:

```
import matplotlib.pyplot as plt
```

Module Initialization

Experis KickStart ManpowerGroup

- A module is loaded and initialized only once, regardless of the number of times you import the module in your script
- The first time a module is loaded into a running Python script, it is initialized by executing the code in the module
- If another module in your code imports the same module again, it will not be loaded again, so local variables inside the module act as a “singleton”

Accessing Modules from Another Directory

Experis KickStart ManpowerGroup

- Modules may be useful for more than one project, and in that case it makes less sense to keep a module in a particular directory that's tied to a specific project
- The Python interpreter searches for the modules files in a list of directories that it receives from the **sys.path** variable
- If you want to use a Python module from a location other than the same directory where your main program is, you have a few options
  - Append the path of the module to the sys.path variable by calling sys.path.append()
  - Copy the module to your main system Python path, which will make the module available system-wide

Accessing Modules from Another Directory

Experis KickStart ManpowerGroup

- For example, let's move ui.py to a different directory, e.g., C:\Python
- When trying to run game.py, we'll get the following error:

```
C:\Users\roi\Anaconda3\python.exe C:/PycharmProjects/mygame/game.py
Traceback (most recent call last):
  File "C:/PycharmProjects/mygame/game.py", line 4, in <module>
    import ui
ModuleNotFoundError: No module named 'ui'

Process finished with exit code 1
```

- To fix this error, you may append the path C:\Python to the system path *before* the import command:

```
import sys
sys.path.append(r"C:\Python")

# import the ui module
import ui
…
```

Accessing Modules from Another Directory

- Another option is to copy ui.pi into the main system Python path
- To find out what path Python checks, run the Python interpreter, import the sys module, and print sys.path:

```
(base) C:\Users\roi>python
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 10:22:32) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> print(sys.path)
['', 'C:\\Users\\roi\\Anaconda3\\python36.zip', 'C:\\Users\\roi\\Anaconda3\\DLLs', 'C:\\Users\\roi\\
Anaconda3\\lib', 'C:\\Users\\roi\\Anaconda3', 'C:\\Users\\roi\\AppData\\Roaming\\Python\\Python36\\s
ite-packages', 'C:\\Users\\roi\\Anaconda3\\lib\\site-packages', 'C:\\Users\\roi\\Anaconda3\\lib\\sit
e-packages\\win32', 'C:\\Users\\roi\\Anaconda3\\lib\\site-packages\\win32\\lib', 'C:\\Users\\roi\\An
aconda3\\lib\\site-packages\\Pythonwin']
```

- For example, we can copy ui.py to C:\Users\%UserName%\Anaconda3\Lib\site-packages
- This will make the module available system-wide
  - For example, you can use this module from the Python shell or IPython Notebook

Exercise (27)

Experis KickStart ManpowerGroup

- Create a module input_helper.py that will provide functions for getting different types of inputs from the user, while handling any possible error in the input, and letting the user fix the errors
- The module should support the following operations:
  - Get an integer from the user
  - Get a positive integer from the user
  - Get an integer in a specified interval [a, b]
  - Get a pair of two integers from the user (e.g., get the coordinates of a point)
- The functions should keep asking the user for input until he/she enters a valid one
- Make this module available system-wide
- Test the functions of this module from a Jupyter Notebook

Exercise (27)

Experis KickStart ManpowerGroup

- Sample run:

```
import input_helper
```

```
x = input_helper.get_positive_int()
x
```

```
Enter a positive integer: hello
Incorrect input. Try again.
Enter a positive integer: -2
Number must be positive. Try again.
Enter a positive integer: 5

5
```

```
p = input_helper.get_pair()
p
```

```
Enter an integer: 8
Enter an integer: 6

(8, 6)
```

Python Packages

Experis KickStart ManpowerGroup

- Packages are the natural way to organize and distribute larger Python projects
- A Python **package** is simply a structured arrangement of modules within a directory on the file system
- To make a package, the module files are placed in a directory, along with a file named **__init__.py**
- This file is run when the package is imported and may perform some initialization and its own imports
- It may be an empty file if no special initialization is required, but it must exist for the directory to be considered by Python to be a package
  - This is done to prevent directories with a common name, such as string, from unintentionally hiding valid modules that occur later on the module search path

Python Packages

- For example, the NumPy package exists as the following directory:

```
numpy/
     __init__.py
     core/
     fft/
          __init__.py
          fftpack.py
          info.py
          ...
     linalg/
          __init__.py
          linalg.py
          info.py
          ...
     polynomial/
          __init__.py
          chebyshev.py
          legendre.py
          ...
     random/
     version.py
     ...
```

Importing Packages

- For example, `polynomial` is a subpackage of the `numpy` package containing several modules, including `legendre`, which may be imported as

```
import numpy.polynomial.legendre
```

- To avoid having to use this full dotted syntax in actually referring to the module's attributes, it is convenient to use

```
from numpy.polynomial import legendre
```

Creating a Package

Experis KickStart ManpowerGroup

- The steps to create a Python package are:
  - Create a directory and give it your package's name
  - Put your modules in it
  - Create a **__init__.py** file in the directory
- For example, let’s create a package for our game
- First, we’ll use the directory C:\PycharmProjects\mygame as our package directory
  - It already contains the modules game.py and ui.py
- Thus, we only need to create an empty file named __init__.py inside mygame
- Now copy the package to C:\Users\%UserName%\Anaconda3\Lib\site-packages
- The package is now available system-wide

Testing the Package

- Create a new Jupyter notebook
- To use the module game, we can import it in two ways:

```
import mygame.game
```

```
mygame.game.start()
```

```
Welcome to my amazing game!
Press ENTER to start
```

```
from mygame import game
```

```
game.start()
```

```
Welcome to my amazing game!
Press ENTER to start
```

- In the first method, we must use the mygame prefix whenever we access the module game
- In the second method, we don't, because we import the module into our module's namespace

Exporting Modules from a Package

- To make a module accessible directly from the package itself, you can import the module in the __init__.py file
- For example:

```
# __init__.py
from . import game  # relative import
```

  - This imports the game module relative to the current package
- Now we only need to import the package in order to work with its modules:

```
import mygame
```

```
mygame.game.start()
```

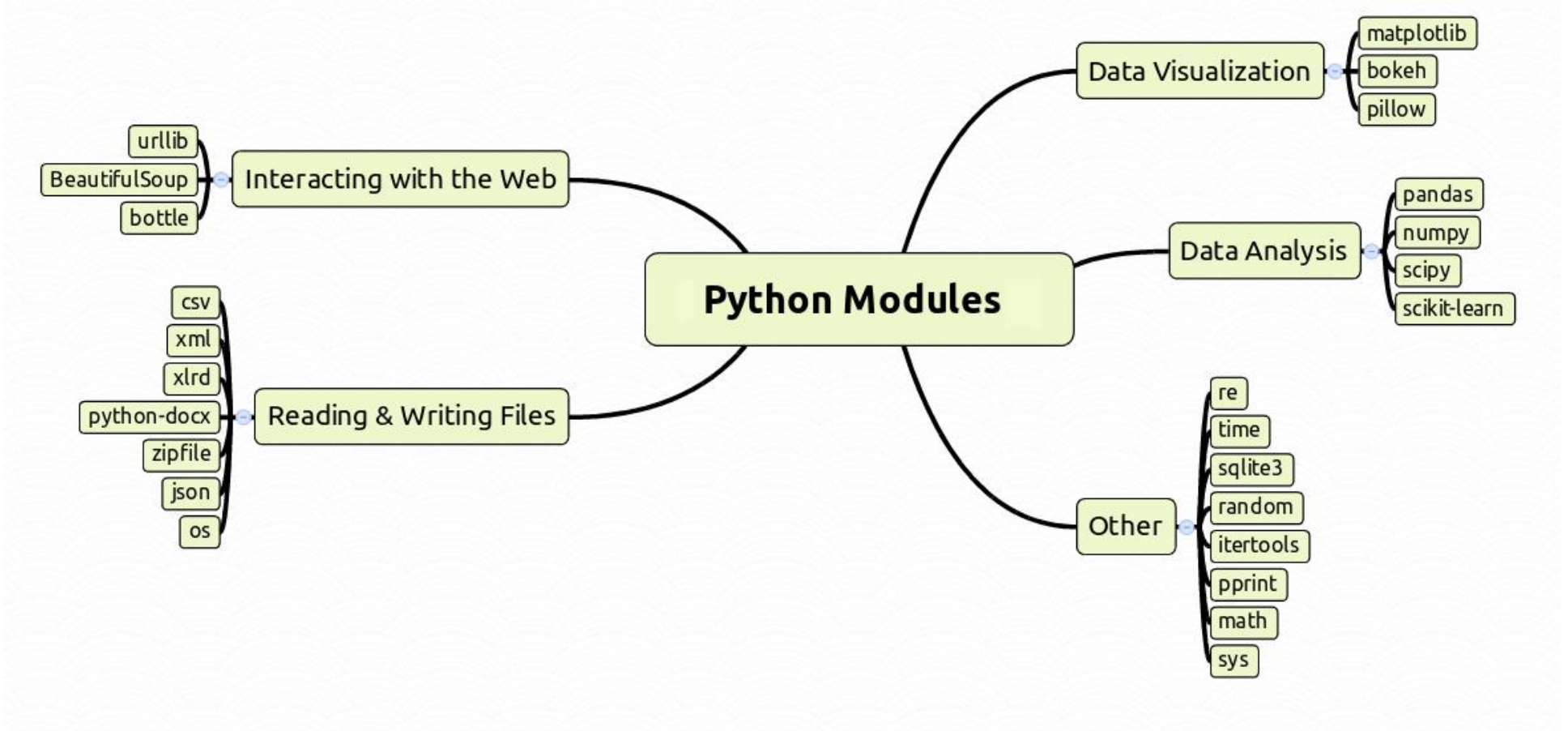
```
Welcome to my amazing game!
Press ENTER to start
```

Summary

Experis KickStart ManpowerGroup

- Use import x for importing packages and modules
- Use from x import y where x is the package prefix and y is the module name
- Use import y as z only when z is a standard abbreviation (e.g., np for numpy)
- Use from x import y as z if two modules named y are to be imported or if y is an inconveniently long name

Python Built-in Modules/Packages



Python Built-in Modules/Packages

Experis KickStart ManpowerGroup

- A list of some of the major, freely available Python modules and packages:

Module/Package	Description
os, sys	Operating system services
math, cmath	Mathematical functions
fractions	Rational number arithmetic
random	Random number generator
collections	Data types for containers that extend the functionality of dictionaries, tuples, etc
itertools	Tools for efficient iterators that extend the functionality of simple Python loops
datetime	Parsing and manipulating dates and times
re	Regular expressions
argparse	Parser for command line options and arguments
urllib	URL (including web pages) opening, reading and parsing
beautifulsoup	HTML parser, with handling of malformed documents
logging	Python's built-in logging module

Python Built-in Modules/Packages

Experis KickStart ManpowerGroup

Module/Package	Description
unittest	Unit testing framework for systematically testing and validating units of code
pdb	The Python debugger
xml, lxml	XML parsers
json	JSON parser
pyodbc	Python SQL driver
pymongo	Python MongoDB driver
visual	Three-dimensional visualization
numpy	Numerical and scientific computing
scipy	Scientific computing algorithms
matplotlib	Plotting
pandas	Data manipulation and analysis with table-like data structures
scikit-learn	Machine learning

The sys Module

Experis KickStart ManpowerGroup

- The sys module provides some useful system-specific parameters and functions
- **sys.argv** is a list of strings, that holds the command line arguments passed to a Python program when it is executed
  - The first item, sys.argv[0], is the name of the program itself
- This allows for a degree of interactivity without having to read from configuration files or require direct user input,
- It also allows other programs or shell scripts to call a Python program and pass it particular input values or settings

The sys Module

- For example, the following script squares a given number:

```
#square.py
import sys

num = int(sys.argv[1])
print(num, "squared is", num**2)
```

- Running this program from the command line with

```
python square.py 3
```

produces the following output:

```
C:\Python\sys>python square.py 3
3 squared is 9
```

- Because we did not hard-code the value of num, the same program can be run with any number

The sys Module

Experis KickStart ManpowerGroup

- **sys.exit()** causes a program to terminate and exit from Python
- This happens “cleanly,” so that any commands specified in a try statement’s finally clause are executed first and any open files are closed
- The optional argument to sys.exit() can be any object:
  - If it is an integer, it is passed to the shell and indicates a success or failure of the program
    - 0 usually denotes “successful” termination of the program and nonzero values indicate some kind of error
  - Passing no argument or None is equivalent to 0
  - If it is a string, it is passed to stderr (the standard error stream)
    - Typically it appears as an error message on the console (unless redirected elsewhere by the shell)

The sys Module

- A common way to help users with scripts that take command line arguments is to issue a usage message if they get it wrong, as in the following code example:

```
#square.py
import sys

try:
    num = int(sys.argv[1])
except (IndexError, ValueError):
    sys.exit("Please enter an integer, <n>, on the command line.\n"
             "Usage: python {} <n>".format(sys.argv[0]))

print(num, "squared is", num**2)
```

```
C:\Python\sys>python square.py
Please enter an integer, <n>, on the command line.
Usage: python square.py <n>

C:\Python\sys>python square.py 5
5 squared is 25
```

The os Module

Experis KickStart ManpowerGroup

- The **os** module provides various operating system interfaces in a platform-independent way
- It provides a number of functions for retrieving information about the context in which the Python process is running
- For example, **os.uname()** returns information about the operating system running Python and the network name of the machine running the process
- **os.getenv(key)** returns the value of the environment variable key if it exists
- Commonly used environment variables:
  - HOME: the path to the user's home directory
  - PWD: the current working directory
  - USER: the current user's username and
  - PATH: the system path environment variable

```
print(os.getenv('HADOOP_HOME'))
```

```
C:\Hadoop
```

File System Commands

Experis KickStart ManpowerGroup

- The os module also provides functions to navigate the system directory tree and manipulate files and directories from within a Python program

Function	Description
os.listdir(*path*='.')	List the entries in the directory given by *path* (or the current working directory if this is not specified)
os.remove(*path*)	Delete the file *path* (raises an OSError if path is a directory; use os.rmdir instead)
os.rmdir(*path*)	Delete the directory *path*. If the directory is not empty, an OSError is raised.
os.rename(*old_name*, *new_name*)	Rename the file or directory *old_name* to *new_name*. If a file with the name new_name already exists, it will be overwritten.
os.mkdir(*path*)	Create the directory named *path*
os.system(*command*)	Execute *command* in a subshell. If the command generates any output, it is redirected to the interpreter standard output stream, stdout.

File System Commands

- Example for listing all the files in a user-supplied folder:

```
dir_name = input("Enter a path: ")
for filename in os.listdir(dir_name):
    print(filename)
```

```
Enter a path: C:\Notebooks\pandas
.ipynb_checkpoints
DataFrame.ipynb
DataFrameBasicsEx.ipynb
DescriptiveStatistics.ipynb
DescriptiveStatisticsEx.ipynb
euro_winners.csv
Indexing.ipynb
IndexingEx.ipynb
```

Pathname Manipulations

Experis KickStart ManpowerGroup

- os.path module provides a number of useful functions for manipulating pathnames:

Function	Description
os.path.basename(*path*)	Return the basename of the pathname *path* giving a relative or absolute path to the file: this usually means the filename.
os.path.dirname(*path*)	Return the directory of the pathname *path*
os.path.split(*path*)	Split *path* into a directory and a filename, returned as a tuple (equivalent to calling dirname and basename) respectively
os.path.splitext(*path*)	Split *path* into a 'root' and an 'extension' (returned as a tuple pair)
os.path.exists(*path*)	Return True if the directory or file path exists, and False otherwise
os.path.getmtime(*path*)	Return the time of last modification of *path*
os.path.getsize(*path*)	Return the size of *path* in bytes
os.path.join(*path1*, *path2*, ...)	Return a pathname formed by joining the path components path1, path2, etc. with the directory separator appropriate to the operating system being used.

Pathname Manipulations

- Some examples referring to a file C:\Notebooks\Test.ipynb:

```
os.path.basename(r'C:\Notebooks\test.ipynb')  # just the filename
```

```
'test.ipynb'
```

```
os.path.dirname(r'C:\Notebooks\test.ipynb')  # just the directory
```

```
'C:\\Notebooks'
```

```
# Directory and filename in a tuple
os.path.split(r'C:\Notebooks\test.ipynb')
```

```
('C:\\Notebooks', 'test.ipynb')
```

```
# File path stem and extension in a tuple
os.path.splitext(r'C:\Notebooks\test.ipynb')
```

```
('C:\\Notebooks\\test', '.ipynb')
```

```
# Join directories and/or file name
os.path.join(r'C:\Notebooks', 'test.ipynb')
```

```
'C:\\Notebooks\\test.ipynb'
```

```
os.path.exists(r'C:\Notebooks\test.py')  # file does not exist
```

```
False
```

Exercise (28)

Experis KickStart ManpowerGroup

- Suppose you have a directory of data files identified by filenames containing a date in the form data-DD-Mon-YY.txt
  - DD is the two-digit day number, Mon is the three-letter month abbreviation and YY is the last two digits of the year, for example ’02-Feb-18’
- Write a program that converts the filenames into the form data-YYYY-MM-DD.txt so that an alphanumeric ordering of the filenames puts them in chronological order

The random Module

Experis KickStart ManpowerGroup

- For simulations, modeling and some numerical algorithms it is often necessary to generate random numbers from some distribution
- Python, as many other programming languages, implements a **pseudorandom number generator** (PRNG) – an algorithm that generates a sequence of numbers that approximates the properties of "truly" random numbers
- Such sequences are determined by an **originating seed** and are always the same following the same seed
  - This is good for reproducing calculations involving random numbers, but a bad thing if used for cryptography, where the random sequence must be kept secret
- Any PRNG will yield a sequence that eventually repeats
  - A good generator will have a long period
- The PRNG implemented by Python is the Mersenne Twister
  - A well-respected and much-studied algorithm with a period of $2^{19937} - 1$

Generating Random Numbers

Experis KickStart ManpowerGroup

- The random number generator can be seeded with any *hashable object*
  - e.g., an immutable object such as an integer
- When the random module is first imported, it is seeded with a representation of the current system time
  - unless the operating system provides a better source of a random seed
- The PRNG can be reseeded at any time with a call to **random.seed()**
- The basic random number method is **random.random()**
  - It generates a random number selected from the uniform distribution in the semi-open interval [0, 1) – that is, including 0 but not including 1

Generating Random Numbers

Experis KickStart
ManpowerGroup

```
import random
random.random()    # PRNG seeded "randomly"
```

```
0.22321073814882275
```

```
random.seed(42)    # seed the PRNG with a fixed value
```

```
random.random()
```

```
0.6394267984578837
```

```
random.random()
```

```
0.025010755222666936
```

```
random.seed(42)    # reseed with the same value as before
```

```
random.random()
```

```
0.6394267984578837
```

```
random.random()
```

```
0.025010755222666936
```

Generating Random Numbers

- To select a random *floating point* number, *N*, from a given range, $a \leq N \leq b$, use **random.uniform**(*a*, *b*):

```
random.uniform(-2, 2)
```

```
-0.899882726523523
```

```
random.uniform(-2, 2)
```

```
-1.107157047404709
```

- To select a random *integer, N*, in a given range, $a \leq N \leq b$, use **random.randint**(*a*, *b*):

```
random.randint(5, 15)
```

```
6
```

```
random.randint(5, 15)
```

```
15
```

Generating Random Numbers

Experis KickStart ManpowerGroup

- The random module has several methods for drawing random numbers from nonuniform distributions (see the documentation)
- For example, to return a number from the normal distribution with mean *mu* and standard deviation *sigma*, use **random.normalvariate**(*mu*, *sigma*):

```
random.normalvariate(100, 15)
```

```
113.62214497887028
```

```
random.normalvariate(100, 15)
```

```
102.40507603179206
```

Random Selections and Permutations

Experis KickStart ManpowerGroup

- **random.choice()** selects an item at random from a sequence such as a list:

```
list1 = [10, 5, 2, -3.4, "hello"]
random.choice(list1)
```

```
10
```

```
random.choice(list1)
```

```
5
```

- **random.shuffle()** randomly shuffles (permutes) the items of the sequence *in place:*

```
random.shuffle(list1)
list1
```

```
['hello', -3.4, 2, 10, 5]
```

  - Because the random permutation is made in place, the sequence must be mutable: you can't, for example, shuffle tuples

Random Sequences

- **random.sample**(*population*, *k*) draws a list of *k* unique elements from a sequence or set (without replacement)

```
lottery_numbers = range(1, 38)
winners = random.sample(lottery_numbers, 6)
winners
```

```
[35, 27, 15, 29, 18, 1]
```

Exercise (29)

Experis KickStart ManpowerGroup

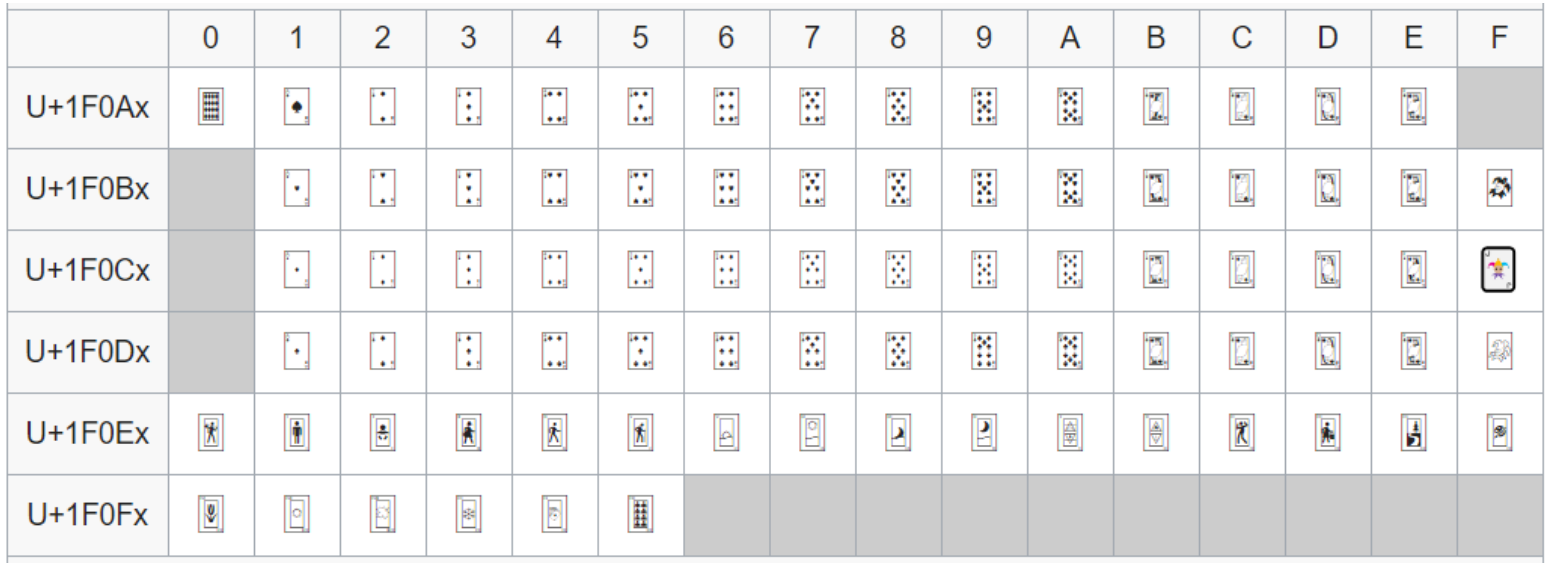
- Define a function **make_deck()** that creates a randomly shuffled deck of cards
- The deck contains 52 cards, 13 of each suit: clubs, diamonds, hearts, and spades
- Each card should be represented as a two-letter string:
  - The first letter indicates the card suite: ['S', 'H', 'D', 'C‘]
  - The second letter indicates the card rank: ['A', '2', '3', '4', '5', '6', '7', '8', '9', 'T', 'J', 'Q', 'K']
  - e.g., king of spades is ”SK”, 5 of clubs is ”C5”, 10 of hearts is “HT”, etc.
- Define another function **deal_hand**(*n*, *deck*) that draws a hand of *n* random cards from *deck*
- Sample run:

```
deck = make_deck()
hand = deal_hand(5, deck)
print(hand)
```

```
['CK', 'S3', 'HJ', 'D3', 'SK']
```

Exercise (29) Cont.

- If time permits: define a function **draw_cards**(cards) that display the cards visually using their Unicode representations:



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+1F0Ax																
U+1F0Bx																
U+1F0Cx																
U+1F0Dx																
U+1F0Ex																
U+1F0Fx																

- Sample usage:

```
deck = make_deck()
hand = deal_hand(5, deck)
print(hand)
draw_cards(hand)
```

```
['S5', 'C5', 'D2', 'H8', 'D9']
```

Exercise (30)

- The *Monty Hall problem* is a famous conundrum in probability which takes the form of a hypothetical game show
- The contestant is presented with three doors: behind one is a car and behind each of the other two is a goat
- The contestant picks a door and then the game show host opens a different door to reveal a goat
- The host knows which door conceals the car
- The contestant is then invited to switch to the other closed door or stick with his or her initial choice
- Write a program in Python that finds the best strategy for winning the car and its probability of winning, by running simulations of the game

The datetime Module

- Python's **datetime** module provides classes for manipulating dates and times:
  - datetime.date – an idealized naïve date (unaware of time zones)
  - datetime.time – an idealized time, independent of any particular day
  - datetime.datetime – a combination of a date and time
  - datetime.timedelta – a duration expressing the difference between two date, time, or datetime instances to microsecond resolution
  - datetime.timezone – a time zone information object
- Objects of these types are immutable

Date Objects

- To create a date object, pass valid year, month and day numbers explicitly or call the date.today() constructor:

```
from datetime import date
```

```
my_date = date(2012, 7, 27)
my_date
```

```
datetime.date(2012, 7, 27)
```

```
today = date.today()
today
```

```
datetime.date(2018, 7, 11)
```

```
date(2018, 6, 31)   # illegal date
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-28-fd44869ace8f> in <module>()
----> 1 date(2018, 6, 31)

ValueError: day is out of range for month
```

- Parsing dates to and from strings is also supported (explained later)

Date Objects

Experis KickStart ManpowerGroup

- Some useful date attributes and methods:

```
my_date.day
```

```
27
```

```
my_date.month
```

```
7
```

```
my_date.year
```

```
2012
```

```
my_date.weekday()    # Monday = 0, Tuesday = 1, ..., Sunday = 6
```

```
4
```

```
my_date.ctime()      # C-standard time output
```

```
'Fri Jul 27 00:00:00 2012'
```

Date Objects

Experis KickStart ManpowerGroup

- Dates can also be compared (chronologically):

```
my_date < today
```

```
True
```

```
my_date == today
```

```
False
```

- Subtracting two dates returns a **datetime.timedelta** object, which represents the difference between the two dates:

```
today - my_date
```

```
datetime.timedelta(2175)
```

- timedelta objects have various attributes for getting the time difference in various units (e.g., days, hours, minutes, etc.)

Time Objects

Experis KickStart ManpowerGroup

- A **datetime.time** object represents a (local) time of day to the nearest microsecond
- To create a time object, pass the number of hours, minutes, seconds and microseconds (missing values default to zero)

```
from datetime import time
```

```
lunchtime = time(12, 30)
lunchtime
```

```
datetime.time(12, 30)
```

```
lunchtime.isoformat()  # HH:MM:SS
```

```
'12:30:00'
```

```
precise_time = time(4, 46, 36, 501982)
precise_time.isoformat()
```

```
'04:46:36.501982'
```

DateTime Objects

- A **datetime.datetime** object contains the information from both the date and time objects: year, month, day, hour, minute, second, microsecond
- You can pass values for these quantities directly to the datetime() constructor, or use the method **now()** to get the current date and time:

```
from datetime import datetime
```

```
now = datetime.now()
now
```

```
datetime.datetime(2018, 7, 29, 6, 2, 4, 400532)
```

```
now.isoformat()
```

```
'2018-07-29T06:02:04.400532'
```

```
now.ctime()
```

```
'Sun Jul 29 06:02:04 2018'
```

Date and Time Formatting

Experis KickStart ManpowerGroup

- date, time and datetime objects support a method **strftime()** to output their values as a string formatted according to the format specifiers listed in following table:

Specifier	Description
%a	Abbreviated weekday (Sun, Mon, etc.)
%A	Full weekday (Sunday, Monday, etc.)
%w	Weekday number (0=Sunday, 1=Monday, ..., 6=Saturday)
%d	Zero-padded day of month: 01, 02, 03, ..., 31
%b	Abbreviated month name (Jan, Feb, etc.)
%B	Full month name (January, February, etc.)
%m	Zero-padded month number: 01, 02, ..., 12
%y	Year without century (two-digit, zero-padded): 01, 02, ..., 99
%Y	Year with century (four-digit, zero-padded): 0001, 0002, ... 9999

Specifier	Description
%H	24-hour clock hour, zero-padded: 00, 01, ..., 23
%I	12-hour clock hour, zero-padded: 00, 01, ..., 12
%p	AM or PM (or locale equivalent)
%M	Minutes (two-digit, zero-padded): 00, 01, ..., 59
%S	Seconds (two-digit, zero-padded): 00, 01, ..., 59
%f	Microseconds (six-digit, zero-padded): 000000, 000001, ..., 999999
%%	The literal % sign

Date and Time Formatting

Experis KickStart ManpowerGroup

- For example:

```
now.strftime("%A, %d %B %Y")
```

```
'Sunday, 29 July 2018'
```

```
now.strftime("%H:%M:%S on %d/%m/%y")
```

```
'06:03:18 on 29/07/18'
```

- The reverse process, parsing a string into a datetime object is done by **strptime()**:

```
queen_birthtime = datetime.strptime("April 21, 1926 02:40", "%B %d, %Y %H:%M")
print(queen_birthtime.strftime("%I:%M %p on %A, %d %b %Y"))
```

```
02:40 AM on Wednesday, 21 Apr 1926
```

Adding Dates and Times

Experis KickStart ManpowerGroup

- Using the **datetime** and **timedelta** objects, you can perform date and time addition/subtraction

```
from datetime import datetime
from datetime import timedelta
```

```
# Add 1 day
print(datetime.now() + timedelta(days=1))
```

```
2018-08-06 08:25:40.147347
```

```
# Subtract 60 seconds
print(datetime.now() - timedelta(seconds=60))
```

```
2018-08-05 08:24:59.764127
```

```
# Pass multiple parameters(1 day and 5 minutes)
print(datetime.now() + timedelta(days=1, minutes=5))
```

```
2018-08-06 08:31:48.009242
```

  - Other parameters that can be passed to timedelta: weeks, days, hours, minutes, seconds, microseconds, milliseconds

Exercise (31)

Experis KickStart ManpowerGroup

- Define a function get_days_until_birthday(*day, month*) that returns the number of days from today until your next birthday as specified by the *day* and *month* arguments
- Sample run:

```
days = get_days_until_birthday(27, 7)
print("{} days are left until your birthday!".format(days))
```

```
356 days are left until your birthday!
```

IPython Magic Functions

Experis KickStart ManpowerGroup

- IPython provides many “magic” functions (or simply *magics)* to speed up coding and experimenting within the IPython shell
- There are two types of magics:
  - **Line magics**: prefixed with %, and their arguments are given on a single line
  - **Cell magics**: prefixed with %%, and act on a series of Python commands
- A list of currently available magic functions can be obtained by typing %lsmagic

IPython Magic Functions

Experis KickStart ManpowerGroup

- Useful IPython line magics:

Magic	Description
%alias	Create an alias to a system command
%alias_magic	Create an alias to an existing IPython magic
%bookmark	Interact with IPython's directory bookmarking system
%cd	Change the current working directory
%edit	Create or edit Python code within a text editor and then execute it
%env	List the system environment variables, such as $HOME
%history	List the input history for this IPython session
%load	Read in code from a provided file and make it available for editing
%recall	Place one or more input lines from the command history at the current input prompt
%rerun	Reexecute previous input from the numbered command history
%reset	Reset the namespace for the current IPython session
%run	Execute a named file as a Python script within the current session

IPython Magic Functions

Experis KickStart
ManpowerGroup

Magic	Description
%save	Save a set of input lines or macro (defined with %macro) to a file with a given name
%sx or !!	Shell execute: run a given shell command and store its output
%timeit	Time the execution of a provided Python statement
%who	Output all the currently defined variables
%who_ls	As for %who, but return the variable names as a list of strings
%whos	As for %who, but provides more information about each variable

Commands History

- To view the history of the IPython commands, use the **%history** or **%hist** magic function
- By default only the entered statements are output
- It is often more useful to output the line numbers as well, which is achieved using the

```
%history -n
```

```
   1:
name = "John"
print(name)
   2: age = 25
   3: In[1]
   4: exec(In[1])
   5: import math
   6: math.sin(2)
   7: math.cos(2)
   8: Out[6] ** 2 + Out[7] ** 2
   9: _
  10: %history -n
```

Commands History

- To output a specific line or range of lines, refer to them by number and/or number range when calling %history:

```
%history -n 2-5
```

```
2: age = 25
3: In[1]
4: exec(In[1])
5: import math
```

- The %history function can also take an additional option: -o, which displays the output as well as the input

Recalling and Rerunning Code

- To reexecute one or more lines from your IPython history, use %rerun with a line number or range of line numbers:

```
In [1]: import math

In [2]: angles = [0, 30, 60, 90]

In [3]: for angle in angles:
            sine_angle = math.sin(math.radians(angle))
            print('sin({:2d}) = {:.5f}'.format(angle, sine_angle))
sin( 0) = 0.00000
sin(30) = 0.50000
sin(60) = 0.86603
sin(90) = 1.00000

In [4]: angles = [15, 45, 75]

In [5]: %rerun 3
=== Executing: ===
for angle in angles:
    sine_angle = math.sin(math.radians(angle))
    print('sin({:2d}) = {:.5f}'.format(angle, sine_angle))
=== Output: ===
sin(15) = 0.25882
sin(45) = 0.70711
sin(75) = 0.96593
```

```
In [6]: %rerun 2-3
=== Executing: ===
angles = [0, 30, 60, 90]
for angle in angles:
    sine_angle = math.sin(math.radians(angle))
    print('sin({:2d}) = {:.5f}'.format(angle, sine_angle))
=== Output: ===
sin( 0) = 0.00000
sin(30) = 0.50000
sin(60) = 0.86603
sin(90) = 1.00000
```

Recalling and Rerunning Code

- If you find yourself reexecuting a series of statements frequently, you can define a named macro to invoke them
- Specify line numbers as before:

```
In [7]: %macro sines 3

Macro `sines` created. To execute, type its name (without quotes).
=== Macro contents: ===
for angle in angles:
    sine_angle = math.sin(math.radians(angle))
    print('sin({:2d}) = {:.5f}'.format(angle, sine_angle))

In [8]: angles = [-45, -30, 0, 30, 45]

In [9]: sines

sin(-45) = -0.70711
sin(-30) = -0.50000
sin( 0) = 0.00000
sin(30) = 0.50000
sin(45) = 0.70711
```

Timing Code Execution

- The IPython magic **%timeit** <statement> times the execution of the *single-line* statement <statement>
- The statement is executed N times in a loop, and each loop is repeated R times
  - N is usually a large, number chosen by IPython to yield meaningful results and R = 7 by default
  - The aim of repeating the execution many times is to allow for variations in speed due to other processes running on the system
- The average time per loop for the best of the R repetitions is reported
- For example, to profile the sorting of a random arrangement of the numbers 1–100:

```
import random
numbers = list(range(1, 100))
random.shuffle(numbers)
%timeit sorted(numbers)
```

```
7.17 µs ± 21 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Timing Code Execution

- You can select *N* and *R* explicitly by passing values to the options -n and -r:

```
%timeit -n 10000 -r 5 sorted(numbers)
```

```
7.36 µs ± 166 ns per loop (mean ± std. dev. of 5 runs, 10000 loops each)
```

- The cell magic **%%timeit** enables one to time a *multiline* block of code
- For example, measuring the time of a naive algorithm that finds the factors of an integer *n*:

```
%%timeit
n = 150
factors = set()
for i in range(1, n + 1):
    if n % i == 0:
        factors.add(i)
```

```
10.5 µs ± 80.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Exercise

Experis KickStart ManpowerGroup

- Improve on the algorithm to find the number of factors of an integer shown on the previous slide by:
  - (a) looping the trial factor, *i*, up to no greater than the square root of *n* (why is it not necessary to test values of *i* greater than this?)
  - (b) using a generator
- Compare the execution speed of these alternatives using the %timeit IPython magic

Loading, Executing and Saving code

Experis KickStart ManpowerGroup

- To load code from an external file into the current IPython session, use

```
%load <filename>
```

  - If you want only certain lines from the input file, specify them after the -r option
  - This magic enters the lines in the notebook, so they can be edited before being executed
- To load *and execute* code from a file, use

```
%run <filename>
```

  - Pass any command line options after *filename*
  - There are a few additional options to %run:
    - **-i**: Run the script in the current IPython namespace instead of an empty one (i.e., the program will have access to variables defined in the current IPython session)
    - **-t**: Output timing information at the end of execution (pass an integer to the additional option **-N** to repeat execution that number of times)

Loading, Executing and Saving code

Experis KickStart ManpowerGroup

- For example, let's create a Python script, codon_lookup.py (saved in C:\Python) that creates a dictionary, codon_table, mapping codons to amino acids where each amino acid is identified by its one-letter abbreviation (e.g., R = arginine)
  - The stop codons, signaling termination of RNA translation, are identified with the single asterisk character *

```
# codon_lookup.py

bases = ['U', 'C', 'A', 'G']
codons = [a+b+c for a in bases for b in bases for c in bases]
amino_acids = 'FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG'
codon_table = dict(zip(codons, amino_acids))
```

- This script can be executed within IPython notebook with %run codon_lookup.py

Loading, executing and saving code

Experis KickStart ManpowerGroup

- For example, let's create a Python script, codon_lookup.py, that creates a dictionary, codon_table, mapping codons to amino acids where each amino acid is identified by its one-letter abbreviation (e.g., R = arginine)
  - The stop codons, signaling termination of RNA translation, are identified with an asterisk *

```
# codon_lookup.py
bases = ['U', 'C', 'A', 'G']
codons = [a+b+c for a in bases for b in bases for c in bases]
amino_acids = 'FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG'
codon_table = dict(zip(codons, amino_acids))
```

- This script can be executed within IPython notebook with %run codon_lookup.py

```
In [1]: %run codon_lookup.py

In [2]: codon_table

Out[2]: {'AAA': 'K',
         'AAC': 'N',
         'AAG': 'K',
         'AAU': 'N',
         'ACA': 'T',
```

Loading, Executing and Saving code

Experis KickStart ManpowerGroup

- To save a range of input lines or a macro to a file, use %save
- Line numbers are specified using the same syntax as %history
- A .py extension is added if you don't add it yourself
- For example, let's define a function to translate an RNA sequence:

```
In [3]: def translate_rna(seq):
            start = seq.find('AUG')
            peptide = []
            i = start
            while i < len(seq) - 2:
                codon = seq[i:i+3]
                a = codon_table[codon]
                if a == '*':
                    break
                i += 3
                peptide.append(a)
            return ''.join(peptide)
```

Loading, Executing and Saving code

- Now, we can save this function to the file translate_rna.py type:

```
In [4]: %save translate_rna 3

The following commands were written to file `translate_rna.py`:
def translate_rna(seq):
    start = seq.find('AUG')
    peptide = []
    i = start
    while i < len(seq) - 2:
        codon = seq[i:i+3]
        a = codon_table[codon]
        if a == '*':
            break
        i += 3
        peptide.append(a)
    return ''.join(peptide)
```

Interacting with the Operating System

Experis KickStart ManpowerGroup

- IPython makes it easy to execute operating system commands from within your notebook
- Any statement preceded by an exclamation mark, !, is sent to the operating system command line (the “system shell”) instead of being executed as a Python statement
- For example, you can list directory contents, delete files, and even execute other programs and scripts

Interacting with the Operating System

```
# return the current working directory
!cd
```

```
C:\Notebooks\Introduction to Python
```

```
# list the files in this directory
!dir
```

```
 Volume in drive C has no label.
 Volume Serial Number is 5CAF-CB37

 Directory of C:\Notebooks\Introduction to Python

04/08/2018  07:22 AM    <DIR>          .
04/08/2018  07:22 AM    <DIR>          ..
04/08/2018  07:09 AM    <DIR>          .ipynb_checkpoints
04/08/2018  06:09 AM             1,193 HelpCommands.ipynb
04/08/2018  07:08 AM             3,909 History.ipynb
04/08/2018  07:21 AM             2,640 InteractingWithOS.ipynb
04/08/2018  05:44 AM             3,282 Markdowns.ipynb
04/08/2018  05:44 AM             1,057 MyFirstNotebook.ipynb
               5 File(s)         12,081 bytes
               3 Dir(s)  30,489,419,776 bytes free
```

```
# write to a new text file
!echo This is a test > test.txt
```

```
!dir
```

```
 Volume in drive C has no label.
 Volume Serial Number is 5CAF-CB37

 Directory of C:\Notebooks\Introduction to Python

04/08/2018  07:26 AM    <DIR>          .
04/08/2018  07:26 AM    <DIR>          ..
04/08/2018  07:09 AM    <DIR>          .ipynb_checkpoints
04/08/2018  06:09 AM             1,193 HelpCommands.ipynb
04/08/2018  07:08 AM             3,909 History.ipynb
04/08/2018  07:23 AM             2,591 InteractingWithOS.ipynb
04/08/2018  05:44 AM             3,282 Markdowns.ipynb
04/08/2018  05:44 AM             1,057 MyFirstNotebook.ipynb
04/08/2018  07:26 AM                17 test.txt
               6 File(s)         12,049 bytes
               3 Dir(s)  30,488,408,064 bytes free
```

```
# display the contents of a file
!type test.txt
```

```
This is a test
```

```
# delete the file
!del test.txt
```

Interacting with the Operating System

- For technical reasons, the cd command must be executed as IPython magic function:

```
%cd C:\Python
```

```
C:\Python
```

```
!dir
```

```
 Volume in drive C has no label.
 Volume Serial Number is 5CAF-CB37

 Directory of C:\Python

02/08/2018  07:34 AM    <DIR>          .
02/08/2018  07:34 AM    <DIR>          ..
02/08/2018  03:25 AM                90 hello.py
02/08/2018  07:31 AM             1,516 MyFirstNotebook.py
02/08/2018  03:32 AM                14 no_output.py
               3 File(s)          1,620 bytes
               2 Dir(s)  30,486,806,528 bytes free
```

Interacting with the Operating System

- You can pass the values of Python variables to operating system commands by prefixing the variable name with a dollar sign, $:

```
python_script = 'hello.py'
!python $python_script
```

```
Life is like riding a bicycle
To keep your balance you must keep moving
```

- The output of a system command can be assigned to a Python variable:

```
script_lines = !type $python_script
```

```
script_lines
```

```
['print("Life is like riding a bicycle")',
 'print("To keep your balance you must keep moving")']
```

  - The output of the command is returned as a list (one item per line)

Aliases

- A system shell command can be given an *alias*: a shortcut for a shell command that can be called as its own magic
- For example, we could define the following alias to list only the directories on the current path:

```
%alias lstdir dir /ad
```

```
%cd C:\Notebooks
%lstdir
```

```
C:\Notebooks
 Volume in drive C has no label.
 Volume Serial Number is 5CAF-CB37

 Directory of C:\Notebooks

04/08/2018  05:41 AM    <DIR>          .
04/08/2018  05:41 AM    <DIR>          ..
02/08/2018  04:16 AM    <DIR>          .ipynb_checkpoints
02/08/2018  06:26 AM    <DIR>          datasets
02/08/2018  06:27 AM    <DIR>          images
05/08/2018  02:51 AM    <DIR>          Introduction to Python
03/08/2018  08:55 AM    <DIR>          Matplotlib
26/07/2018  01:50 AM    <DIR>          MongoDB
```

Exercise (32)

- From Jupyter Notebook:
  - Create a directory C:\temp_dir
  - Change current directory to C:\temp_dir
  - Create the files output.txt and readme.txt in the current directory
  - List all the text files in the current directory
  - Delete the directory C:\temp_dir
  - Open the Windows calculator app (hint: use the calc command)

Python Coding Style

Experis KickStart ManpowerGroup

- The officially recommended coding conventions for Python are provided by a document known as PEP8 (http://www.python.org/dev/peps/pep-0008/)
- Use *four spaces* per indentation level (and never tabs)
- In assignments, put spaces around the = sign
  - For example, a = 10, not a=10
- Separate operators from their operands with single spaces unless operations with different priorities are being combined
  - For example, write x = x + 5, but r2 = x**2 + y**2
- Don't use spaces around the = in keyword arguments
  - For example, in function calls use foo(b=4.5) not foo(b = 4.5)
- Separate top-level function and class definitions by two blank lines; within a class, separate them by one blank line

Python Coding Style

Experis KickStart ManpowerGroup

- Use a maximum of 80 characters per line, where you need to split a line of code over more than one line:
  - Favor implicit line continuation inside parentheses over the explicit use of the character \
  - In arithmetic expressions, break around binary operators so that the new line is *after* the operator
  - As far as possible, line up code so that expressions within parentheses line up, e.g.:

```
#poor style
lengthy_calculation = margin * margin_px + (border * border_px\
                        + padding * padding_px)
# better
lengthy_calculation = (margin * margin_px + (border * border_px +
                                             padding * padding_px))
```

- Avoid putting more than one statement on the same line separated by semicolons
  - For example, instead of a = 1; b = 2, write a, b = 1, 2
- Avoid wildcard imports (from foo import *)

Python Coding Style

Experis KickStart ManpowerGroup

- Functions, modules and packages should have short, all-lowercase names
- Use underscores in function and module names if necessary, but avoid them in package names
- Class names should be in CamelCase, also known as CapWords
  - For example, AminoAcid, not amino_acid
- Define constants in all-capitals with underscores separating words
  - For example, MAX_LINE_LENGTH