

Unit 5

Exceptions

Catching exceptions

- Whenever a runtime error occurs, it creates an exception object.
- The program stops running at this point and Python prints out the traceback, which ends with a message describing the exception that occurred.

For example, dividing by zero creates an exception:

```
>>> print(55/0)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Catching exceptions

So does accessing a non-existent list item:

```
>>> a = []
```

```
>>> print(a[5])
```

```
Traceback (most recent call last):
```

```
  File "<interactive input>", line 1, in <module>
```

```
IndexError: list index out of range
```

Catching exceptions

So does accessing a non-existent list item:

```
>>> a = []
```

```
>>> print(a[5])
```

```
Traceback (most recent call last):
```

```
  File "<interactive input>", line 1, in <module>
```

```
IndexError: list index out of range
```

Catching exceptions

- In each case, the error message on the last line has two parts:
 - The **type of error** before the colon, and
 - **Specifics about the error** after the colon.
- Sometimes **we want to execute an operation** that might cause an exception, but we don't want the program to stop.
- We **can handle** the exception using the **try statement** to “wrap” a region of code.

Catching exceptions

➤ Example: Trying to open a file that do not exist

```
filename = input("Enter a file name: ")
try:
    f = open(filename, "r")
except:
    print("There is no file named", filename)
```

```
Enter a file name: rr
There is no file named rr
```

Catching exceptions

The try statement has **three separate clauses**:

1. try ...
2. except ...
3. finally.

➤ Either the **except or the finally clauses** can be **omitted**, so the above code considers the most common version of the try statement first.

➤ The try statement executes and monitors the statements in the first block.

➤ If **no exceptions occur**, it skips the block under the except clause.

➤ If **any exception occurs**, it executes the statements in the except clause and then continues.

Catching exceptions

```
def exists(filename):  
    try:  
        f = open(filename)  
        f.close()  
        return True  
    except:  
        return False
```

```
filename = input("Enter a file name: ")  
print(exists(filename))
```

```
Enter a file name: rr  
False
```


Catching exceptions

- The function we've just shown is not one we'd recommend.
 - It **opens and closes the file**, which is semantically different from asking “does it exist?”.

How?

- **Firstly**, it might update some timestamps on the file.
- **Secondly**, it might tell us that there is no such file if **some other program already happens** to have the file open, or if our **permission settings don't allow** us to open the file.
- Python provides a **module** called **os.path** within the **os module**.
 - It provides a number of useful functions to **work with paths, files and directories**,

Catching exceptions

```
import os
```

```
# This is the preferred way to check if a file exists.
```

```
if os.path.isfile("c:/temp/testdata.txt"):
```

```
...
```

➤ We can **use multiple except clauses** to handle different kinds of exceptions.

So the program could do one thing if the file does not exist, but do something else if the file was in use by another program.

Raising our own exceptions

- If our program detects an error condition, we can raise an exception.

```
def get_age():
    age = int(input("Please enter your age: "))
    if age < 0:
        # Create a new instance of an exception
        my_error = ValueError("{0} is not a valid age".format(age))
        raise my_error
    return age

print(get_age())
```

```
Please enter your age: -2
Traceback (most recent call last):
  File "C:/Users/Admin/AppData/Local/Programs/Python/Python35/except3.py", line
9, in <module>
    print(get_age())
  File "C:/Users/Admin/AppData/Local/Programs/Python/Python35/except3.py", line
6, in get_age
    raise my_error
ValueError: -2 is not a valid age
>>>
= RESTART: C:/Users/Admin/AppData/Local/Programs/Python/Python35/except3.py =
Please enter your age: 5
5
```

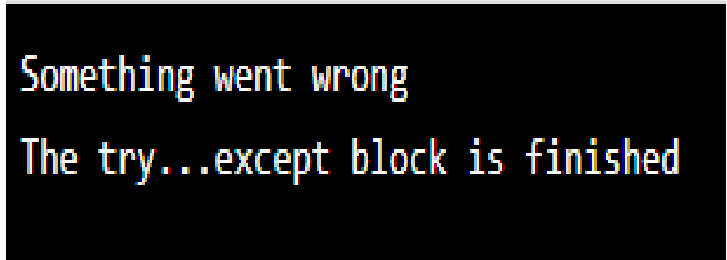
Raising our own exceptions

- **ValueError** is one of the **built-in exception types** which most closely matches the kind of error we want to raise.
- The complete listing of built-in exceptions can be found at the Built-in Exceptions section of the Python Library Reference , again by Python's creator, Guido van Rossum.
- The **error message** includes the **exception type** and the **additional information** that was provided when the exception object was first created.

The finally clause of the try statement

- The finally block will always be executed, no matter if the try block raises an error or not:

```
try:
    x > 3
except:
    print("Something went wrong")
else:
    print("Nothing went wrong")
finally:
    print("The try...except block is finished")
```



```
Something went wrong
The try...except block is finished
```

Strings

A compound data type

- So far we have seen **built-in types** like `int`, `float`, `bool`, `str` and we've seen lists and pairs.
- **Strings, lists, and pairs** are qualitatively different from the others because they are made up of **smaller pieces**.
- In the case of strings, they're **made up of** smaller strings each containing **one character**.
- Types that comprise smaller pieces are called **compound data types**

Working with strings as single things

- A string is also an **object**. So each string instance **has its own attributes and methods**.

```
>>> ss = "Hello, World!"  
>>> tt = ss.upper()  
>>> tt  
'HELLO, WORLD!'
```


Working with strings as single things

```
text = 'PythOn ProGRamming'

print("\nConverted String:")
print(text.upper())

print("\nConverted String:")
print(text.lower())

|
print("\nConverted String:")
print(text.title())

# original string never changes
print("\nOriginal String")
print(text)
```

Converted String:
PYTHON PROGRAMMING

Converted String:
python programming

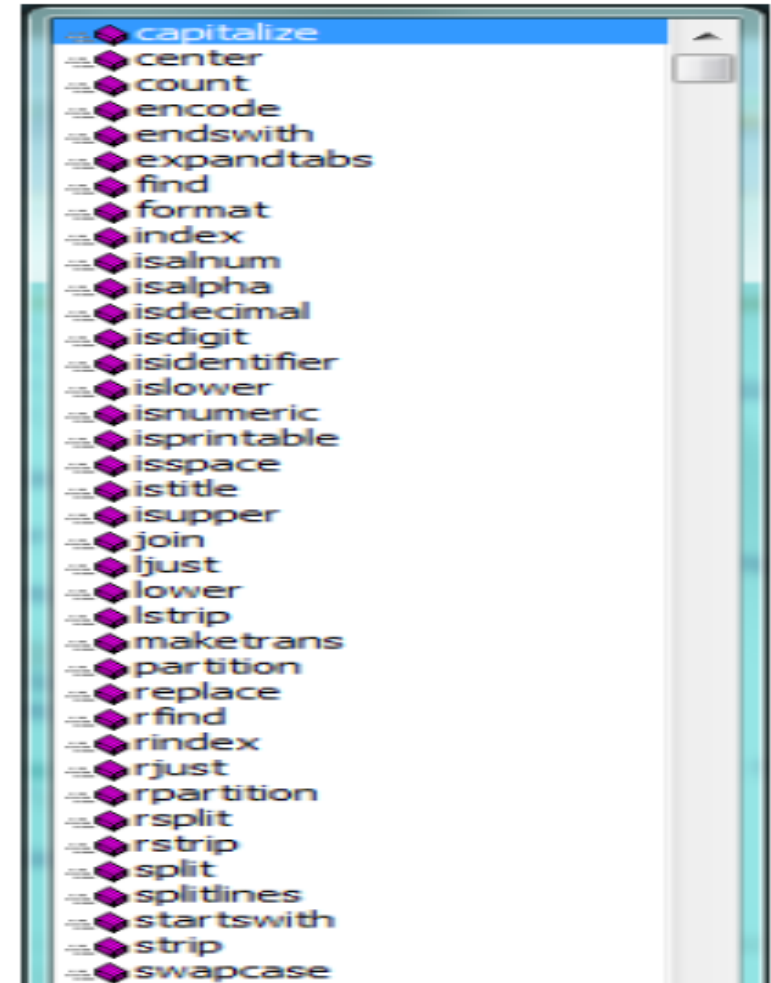
Converted String:
Python Programming

Original String
PythOn ProGRamming

Working with strings as single things

```
ss = "Hello, World!"
```

```
tt = ss.
```



Working with strings as single things

- When you **type the name of the method**, some further help about its **parameter and return type**, and its docstring, will be displayed.
- This is a good example of a tool **PyScripter** using the **meta-information** the docstrings provided by the module programmers.

```
greet = "Hello, World"  
xx= greet.swapcase()  
print(xx)
```

```
** No/Unknown parameters **
```

```
S.swapcase() -> str
```

```
Return a copy of S with uppercase characters converted to lowercase  
and vice versa.
```

Working with the parts of a string

- The **indexing operator** (Python uses square brackets to enclose the index) selects a single character substring from a string:

```
fruit = "banana"
```

```
m = fruit[1]
```

```
print(m)
```

Working with the parts of a string

- We can use **enumerate** to visualize the indices:

```
>>> fruit = "banana"
>>> list(enumerate(fruit))
[(0, 'b'), (1, 'a'), (2, 'n'), (3, 'a'), (4, 'n'), (5, 'a')]
```

```
>>> friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
>>> friends[3]
'Angelina'
```

Length

- The **len function**, when applied to a string, returns the number of characters in a string:

```
>>> fruit = "banana"
>>> len(fruit)
6
```

- To **get the last letter** of a string, you might be tempted to try something like this:

```
sz = len(fruit)
last = fruit[sz]
```

Length

- Alternatively, we can use **negative indices**, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

```
fruit = "banana 100"  
print(fruit[-1])  
print(fruit[-5])  
print(fruit[-20])
```

```
0
```

```
a
```

```
Traceback (most recent call last):
```

```
  File "C:/Users/Admin/AppData/Local/Programs/Python/Python35/String2.py", line  
7, in <module>
```

```
    print(fruit[-20])
```

```
IndexError: string index out of range
```

Traversal and the for loop

- A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called **a traversal**.

```
fruit = "banana 100"  
ix = 0  
while ix < len(fruit):  
    letter = fruit[ix]  
    print(letter)  
    ix += 1
```

b
a
n
a
n
a

1
0
0

Traversal and the for loop

```
fruit = "banana 100"  
  
for ix in fruit:  
    print(ix)
```

b
a
n
a
n
a

1
0
0

Concatenating Two Strings

```
prefixes = "JKLMNOPQ"  
suffix = "ack"  
  
for p in prefixes:  
    print(p + suffix)
```

Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack

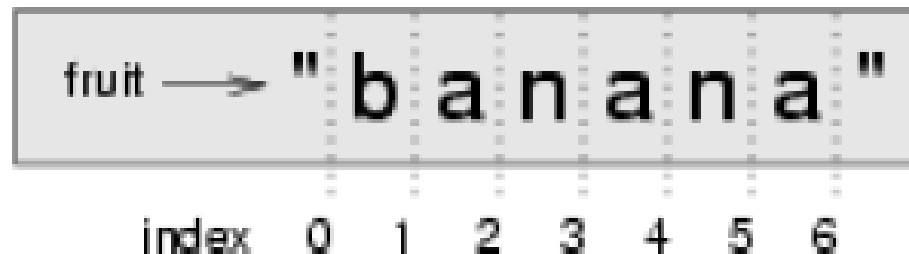
Slices

- A substring of a string is obtained by taking a slice. Similarly, we can slice a list to refer to some sublist of the items in the list:

```
>>> s = "Pirates of the Caribbean"
>>> print(s[0:7])
Pirates
>>> print(s[11:14])
the
>>> print(s[15:24])
Caribbean
>>> friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
>>> print(friends[2:4])
['Brad', 'Angelina']
```

Slices

- The **operator** `[n:m]` returns the part of the string from the **n'th character to the m'th character**, including the first but excluding the last.
- This behaviour makes sense if you imagine the indices pointing between the characters, as in the following diagram:



Slices

- **Three tricks** are added to this:
 - a. If you omit the first index (before the colon), the slice starts at the beginning of the string (or list).
 - b. If you omit the second index, the slice extends to the end of the string (or list).
 - c. Similarly, if you provide value for n that is bigger than the length of the string (or list), the slice will take all the values up to the end. (It won't give an “out of range” error like the normal indexing operation does.)

String comparison

- The comparison operators work on strings. To see if two strings are equal:

```
if word == "banana":  
    print("Yes, we have no bananas!")
```

- Other comparison operations are useful for putting words in lexicographical order:

```
if word < "banana":  
    print("Your word, " + word + ", comes before banana.")  
elif word > "banana":  
    print("Your word, " + word + ", comes after banana.")  
else:  
    print("Yes, we have no bananas!")
```

String comparison

- This is similar to the alphabetical order you would use with a dictionary, except that all the uppercase letters come before all the lowercase letters.

Strings are immutable

- It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
greeting = "Hello, world!"  
greeting[0] = 'J'           # ERROR!  
print(greeting)
```

`TypeError: 'str' object does not support item assignment.`

Strings are immutable

- The solution here is to concatenate a new first letter onto a slice of greeting. This operation has no effect on the original string.

```
greeting = "Hello, world!"  
new_greeting = "J" + greeting[1:]  
print(new_greeting)
```

The in and not in operators

- The in operator tests for membership. When both of the arguments to in are strings, in checks whether the left argument is a substring of the right argument.

```
>>> "p" in "apple"
True
>>> "i" in "apple"
False
>>> "ap" in "apple"
True
>>> "pa" in "apple"
False
```

The in and not in operators

- Note that a string is a substring of itself, and the empty string is a substring of any other string. (Also note that computer scientists like to think about these edge cases quite carefully!)

```
>>> "a" in "a"
```

```
True
```

```
>>> "apple" in "apple"
```

```
True
```

```
>>> "" in "a"
```

```
True
```

```
>>> "" in "apple"
```

```
True
```

The in and not in operators

- The not in operator returns the logical opposite results of in:

```
>>> "x" not in "apple"  
True
```

- Combining the in operator with string concatenation using +, we can write a function that removes all the vowels from a string:

```
def remove_vowels(s):  
    vowels = "aeiouAEIOU"  
    s_sans_vowels = ""  
    for x in s:  
        if x not in vowels:  
            s_sans_vowels += x  
    return s_sans_vowels  
  
test(remove_vowels("compsci") == "cmpsc")  
test(remove_vowels("aAbEefIijOopUus") == "bfjps")
```

A find function

- What does the following function do?

```
def find(strng, ch):  
    """  
        Find and return the index of ch in strng.  
        Return -1 if ch does not occur in strng.  
    """  
    ix = 0  
    while ix < len(strng):  
        if strng[ix] == ch:  
            return ix  
        ix += 1  
    return -1
```

```
test(find("Compsci", "p") == 3)  
test(find("Compsci", "C") == 0)  
test(find("Compsci", "i") == 6)  
test(find("Compsci", "x") == -1)
```

Looping and counting

- The following program counts the number of times the letter a appears in a string, and is another example of the counter pattern introduced in Counting digits:

```
def count_a(text):  
    count = 0  
    for c in text:  
        if c == "a":  
            count += 1  
    return count  
  
test(count_a("banana") == 3)
```

Optional parameters

- To find the locations of the second or third occurrence of a character in a string, we can modify the find function, adding a third parameter for the starting position in the search string:

```
def find2(strng, ch, start):  
    ix = start  
    while ix < len(strng):  
        if strng[ix] == ch:  
            return ix  
        ix += 1  
    return -1  
  
test(find2("banana", "a", 2) == 3)
```

Optional parameters

- The call `find("banana", "a", 2)` now returns 3, the index of the first occurrence of “a” in “banana” starting the search at index 2. What does `find("banana", "n", 3)` return?

```
def find(strng, ch, start=0):  
    ix = start  
    while ix < len(strng):  
        if strng[ix] == ch:  
            return ix  
        ix += 1  
    return -1
```


Optional parameters

- When a function has an optional parameter, the caller may provide a matching argument.
- If the third argument is provided to `find`, it gets assigned to `start`.
- But if the caller leaves the argument out, then `start` is given a default value indicated by the assignment `start=0` in the function definition.
- So the call `find("banana", "a", 2)` to this version of `find` behaves just like `find2`, while in the call `find("banana", "a")`, `start` will be set to the default value of 0.

Optional parameters

- Adding another optional parameter to find makes it search from a starting position, up to but not including the end position:

```
def find(strng, ch, start=0, end=None):  
    ix = start  
    if end is None:  
        end = len(strng)  
    while ix < end:  
        if strng[ix] == ch:  
            return ix  
        ix += 1  
    return -1
```

Optional parameters

- The optional value for end is interesting: we give it a default value None if the caller does not supply any argument.
- In the body of the function we test what end is, and if the caller did not supply any argument, we reassign end to be the length of the string.
- If the caller has supplied an argument for end, however, the caller's value will be used in the loop.

Optional parameters

- The semantics of start and end in this function are precisely the same as they are in the range function.
- Here are some test cases that should pass:

```
ss = "Python strings have some interesting methods."  
test(find(ss, "s") == 7)  
test(find(ss, "s", 7) == 7)  
test(find(ss, "s", 8) == 13)  
test(find(ss, "s", 8, 13) == -1)  
test(find(ss, ".") == len(ss)-1)
```

The built-in find method

- Now that we've done all this work to write a powerful find function, we can reveal that strings already have their own built-in find method. It can do everything that our code can do, and more!

```
test(ss.find("s") == 7)
test(ss.find("s", 7) == 7)
test(ss.find("s", 8) == 13)
test(ss.find("s", 8, 13) == -1)
test(ss.find(".") == len(ss)-1)
```

The built-in find method

- The built-in find method is more general than our version. It can find substrings, not just single characters:

```
>>> "banana".find("nan")
```

```
2
```

```
>>> "banana".find("na", 3)
```

```
4
```

The split method

- It splits a single multi-word string into a list of individual words, removing all the whitespace between them. (Whitespace means any tabs, newlines, or spaces.)

```
>>> ss = "Well I never did said Alice"  
>>> wds = ss.split()  
>>> wds  
['Well', 'I', 'never', 'did', 'said', 'Alice']
```

Cleaning up your strings

- Strings are immutable, so we cannot change the string with the punctuation — we need to traverse the original string and create a new string, omitting any punctuation:

```
punctuation = "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~"
```

```
def remove_punctuation(s):  
    s_sans_punct = ""  
    for letter in s:  
        if letter not in punctuation:  
            s_sans_punct += letter  
    return s_sans_punct
```


Cleaning up your strings

```
import string

def remove_punctuation(s):
    s_sans_punct = ""
    for letter in s:
        if letter not in string.punctuation:
            s_sans_punct += letter
    return s_sans_punct

print(remove_punctuation("Well, I never $ & did!") == "Well I never did said Alice")
```

Cleaning up your strings

- Composing together this function and the split method
- we'll clean out the punctuation, and split will clean out the newlines and tabs while turning the string into a list of words:

Cleaning up your strings

```
import string

def remove_punctuation(s):
    s_sans_punct = ""
    for letter in s:
        if letter not in string.punctuation:
            s_sans_punct += letter
    return s_sans_punct

my_story = """
Pythons are constrictors, which means that they will 'squeeze' the life
out of their prey. They coil themselves around their prey and with
each breath the creature takes the snake will squeeze a little tighter
until they stop breathing completely. Once the heart stops the prey
is swallowed whole. The entire animal is digested in the snake's
stomach except for fur or feathers. What do you think happens to the fur,
feathers, beaks, and eggshells? The 'extra stuff' gets passed out as ---
you guessed it --- snake POOP! """

wds = remove_punctuation(my_story).split()
print(wds)
```

Output

```
['Pythons', 'are', 'constrictors', 'which', 'means', 'that', 'they', 'will', 's  
squeeze', 'the', 'life', 'out', 'of', 'their', 'prey', 'They', 'coil', 'themselv  
es', 'around', 'their', 'prey', 'and', 'with', 'each', 'breath', 'the', 'creatur  
e', 'takes', 'the', 'snake', 'will', 'squeeze', 'a', 'little', 'tighter', 'until  
, 'they', 'stop', 'breathing', 'completely', 'Once', 'the', 'heart', 'stops', '  
the', 'prey', 'is', 'swallowed', 'whole', 'The', 'entire', 'animal', 'is', 'dige  
sted', 'in', 'the', 'snake's', 'stomach', 'except', 'for', 'fur', 'or', 'feather  
s', 'What', 'do', 'you', 'think', 'happens', 'to', 'the', 'fur', 'feathers', 'be  
aks', 'and', 'eggshells', 'The', 'extra', 'stuff', 'gets', 'passed', 'out', 'a  
s', 'you', 'guessed', 'it', 'snake', 'POOP']
```

The string format method

- The easiest and most powerful way to format a string in Python 3 is to use the format method.

```
s1 = "His name is {0}!".format("Arthur")  
print(s1)
```

```
name = "Alice"  
age = 10  
s2 = "I am {1} and I am {0} years old.".format(age, name)
```

The string format method

```
print(s2)
```

```
n1 = 4
```

```
n2 = 5
```

```
s3 = "2**10 = {0} and {1} * {2} = {3:f}".format(2**10, n1, n2, n1 * n2)
```

```
print(s3)
```

Running the script produces:

```
His name is Arthur!
```

```
I am Alice and I am 10 years old.
```

```
2**10 = 1024 and 4 * 5 = 20.000000
```

The string format method

- Each of the replacement fields can also contain a format specification it is always introduced by the : symbol
- This modifies how the substitutions are made into the template, and can control things like:
 - ✓ whether the field is aligned to the left <, center ^, or right >
 - ✓ the width allocated to the field within the result string (a number like 10)
 - ✓ the type of conversion
 - ✓ if the type conversion is a float, you can also specify how many decimal places are wanted (typically, .2f is useful for working with currencies to two decimal places.)

The string format method

```
n1 = "Paris"
```

```
n2 = "Whitney"
```

```
n3 = "Hilton"
```

```
print("Pi to three decimal places is {0:.3f}".format(3.1415926))
```

```
print("123456789 123456789 123456789 123456789 123456789 123456789")
```

```
print("|||{0:<15}|||{1:^15}|||{2:>15}|||Born in {3}|||"  
      .format(n1,n2,n3,1981))
```

```
print("The decimal value {0} converts to hex value {0:x}"  
      .format(123456))
```


The string format method

Pi to three decimal places is 3.142

123456789 123456789 123456789 123456789 123456789 123456789

|||Paris ||| Whitney ||| Hilton|||Born in 1981|||

The decimal value 123456 converts to hex value 1e240

The string format method

You can have multiple placeholders indexing the same argument, or perhaps even have extra arguments that are not referenced at all:

```
letter = """
Dear {0} {2}.
    {0}, I have an interesting money-making proposition for you!
    If you deposit $10 million into my bank account, I can
    double your money ...
"""

print(letter.format("Paris", "Whitney", "Hilton"))
print(letter.format("Bill", "Henry", "Gates"))
```

The string format method

This produces the following:

```
Dear Paris Hilton.
```

```
Paris, I have an interesting money-making proposition for you!  
If you deposit $10 million into my bank account, I can  
double your money ...
```

```
Dear Bill Gates.
```

```
Bill, I have an interesting money-making proposition for you!  
If you deposit $10 million into my bank account I can  
double your money ...
```