INTERMEDIATE TOPICS, CLI AND EXCEPTIONS

CHRISTOPH PAULIK

LIST AND DICT COMPREHENSION

- for creation of lists or dictionaries based on some loop
- shorter than a classic for loop

```
dataset1 = [1.73, 80, 2.4122, -4]
threshold = 2.
result = [x > threshold for x in dataset1]
print(result)
```

```
>>> >>> [False, True, True, False]
```

DICTIONARIES

- we can also construct a dictionary

```
cl = {True: 'larger', False: 'smaller'}
result = {k: cl[k>threshold] for k in dataset1}
print result
```

```
>>> {80: 'larger', 1.73: 'smaller', -4: 'smaller', 2.4122: 'larger'}
```

MULTIPLE ARGUMENTS

```
data = ['a', 'b', 'c', 'd']
result = {i: x for i,x in enumerate(data)}
print result
```

```
{0: 'a', 1: 'b', 2: 'c', 3: 'd'}
```

~~MINI~~ TINY EXERCISE

```
#########################################################
# Tiny Exercise

# Create a list that contains the square value of every
# element in dataset1 using list comprehension.
#########################################################
```

ADVANCED FUNCTION ARGUMENTS

Arguments to functions given as lists or tuples can be unpacked by python

```
def multi(a, b):
    """Documentation does not fit on slide"""
    return a * b

print(multi(2,3))
numbers = [3, 4]
print(multi(*numbers))
```

```
... ... >>> 6
>>> 12
```

The `*` tells python to unpack the arguments

```
print multi(numbers)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: multi() takes exactly 2 arguments (1 given)
```

KEYWORDS

```
def multikw(arg1="dummy", arg2="text"):
    """Documentation does not fit on slide"""
    return " ".join([arg1, arg2])

kwargs = {'arg1': "this text comes from",
          'arg2': "a keyword dictionary"}

print(multikw())
print(multikw(**kwargs))
```

```
dummy text
this text comes from a keyword dictionary
```

* AND ** IN FUNCTION DEFINITIONS

`*` will be a tuple, `**` a dictionary

```
def multi(*args):
    """
    Multiplies all given numbers
    """
    print(type(args))
    print("{} Arguments to multiply".format(len(args)))
    res = 1
    for arg in args:
        res *= arg
    return res
print(multi(2,3,4,5,6))
```

```
<type 'tuple'>
5 Arguments to multiply
720
```

**

```
def print_kw(**kwargs):
    """print keywords"""

    print(type(kwargs))
    for key in kwargs:
        print("{}: {}".format(key, kwargs[key]))

print_kw(argument1=45, argument2="string", test="hello")
```

```
<type 'dict'>
test: hello
argument2: string
argument1: 45
```

FUNCTIONS EVERYWHERE

- Every object can be passed into a function
- e.g. another function

```
def do(f, a, b):
    print f.__doc__ # this is the docstring of the function
    return f(a,b)
def add(a, b):
    """addition"""
    return a+b
def sub(a, b):
    """subtraction"""
    return a-b

print(do(add, 2, 3))
print(do(sub, 3, 2))
```

```
addition
5
subtraction
1
```

CONTROL STRUCTURES

break, continue for–else finally

PASS AND ASSERT

- pass does nothing
- assert checks something

```
name = 2
assert type(name) == str, "name should be a string"
def function():
    pass # can be useful when planning program structure
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AssertionError: name should be a string
```

FOR LOOPS WITH BREAK AND ELSE

```
for n in range(2, 8): # lets debug through this
    for x in range(2, n):
        if n % x == 0:
            print n, 'equals', x, '*', n / x
            break # breaks out of (ends) current loop
    else:
        # loop fell through without finding a factor
        print n, 'is a prime number'
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
```

FOR LOOPS AND CONTINUE

```
for num in range(2, 8):
    if num % 2 == 0: # percent sign is modulo
        print "Found an even number", num
        continue # continue with the next iteration of the loop
    print "Found a number", num
```

```
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
```

COMMAND LINE INTERFACE (CLI)

ARGUMENTS TO YOUR SCRIPTS

Stored in `sys.argv`

contents of file `cli_1.py`

```
if __name__ == '__main__':
    import sys
    print sys.argv
```

```
python cli_1.py test -m hello
```

```
['cli_1.py', 'test', '-m', 'hello']
```

ARGPARSE

official tutorial

contents of file `cli_2.py`

```
import argparse
if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description="get the given name, optionally also the surname")
    parser.add_argument("given_name", help="given name of the person")
    parser.add_argument(
        "-s", "--surname", help="the surname of the person")
    args = parser.parse_args()
    print(args.given_name)
    if args.surname:
        print(args.surname)
```

GETTING USER INPUT

contents of file `user_input.py`

```
num = raw_input("Give me a number: ")
print(type(num))
num = float(num)
print("This is the number you have given me: {:.2f}".format(num))
```

What happens if we do not give a number?

EXCEPTIONS

Are raised when something goes wrong. But can be catched.

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print "division by zero!"
    else:
        print "result is", result
    finally:
        print "executing finally clause"
divide(2, 1)
divide(2, 0)
```

```
result is 2
executing finally clause
division by zero!
executing finally clause
```

```python
print(divide("2", "1"))
```

```
Traceback (most recent call last):
  File "<stdin>", line 12, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

MINI EXERCISE

```
############################################################
# Mini Exercise

# write a function that makes sure that the input is
# a number
############################################################
```