

Python Fundamentals

Handling exceptions

Austin Bingham
🐦 @austin_bingham
austin@sixty-north.com



Presenter

Robert Smallshire
🐦 @robsmallshire
rob@sixty-north.com



pluralsight 
hardcore developer training

Exception Handling

Exception handling is a mechanism for stopping "normal" program flow and continuing at some surrounding context or code block.

Exceptions: Key Concepts

Raise an exception to interrupt program flow.

Exceptions: Key Concepts

Raise an exception to interrupt program flow.

Handle an exception to resume control.

Exceptions: Key Concepts

Raise an exception to interrupt program flow.

Handle an exception to resume control.

Unhandled exceptions will terminate the program.

Exceptions: Key Concepts

Raise an exception to interrupt program flow.

Handle an exception to resume control.

Unhandled exceptions will terminate the program.

Exception objects contain information about the exceptional event.

Similar to other imperative languages

C++



What is exceptional?

Normal



Meltdown!

What is exceptional?

Normal



Meltdown!



```
'''A module for demonstrating exceptions.'''
```

```
def convert(s):  
    '''Convert to an integer.'''  
    x = int(s)  
    return x
```

```
'''A module for demonstrating exceptions.'''
```

```
def convert(s):
```

```
    '''Convert to an integer.'''
```

```
    x = int(s)
```

```
    return x
```

REPL

convert()

int()

ValueError

REPL

convert()

ValueError

REPL



ValueError

```
'''A module for demonstrating exceptions.'''
```

```
def convert(s):  
    '''Convert to an integer.'''  
    try:  
        x = int(s)  
    except ValueError:  
        x = -1  
    return x
```

```
'''A module for demonstrating exceptions.'''
```

```
def convert(s):
```

```
    '''Convert to an integer.'''
```

```
    try:
```

```
        x = int(s)
```

```
    except ValueError:
```

```
        x = -1
```

```
    return x
```



```
'''A module for demonstrating exceptions.'''
```

```
def convert(s):
```

```
    '''Convert to an integer.'''
```

```
    try:
```

```
        x = int(s)
```

```
    except ValueError:
```

```
        x = -1
```

```
    return x
```

```
'''A module for demonstrating exceptions.'''
```

```
def convert(s):  
    '''Convert to an integer.'''  
    try:  
        x = int(s)  
        print("Conversion succeeded! x =", x)  
    except ValueError:  
        print("Conversion failed!")  
        x = -1  
    return x
```

```
'''A module for demonstrating exceptions.'''
```

```
def convert(s):  
    '''Convert to an integer.'''  
    try:  
        x = int(s)  
        print("Conversion succeeded! x =", x)  
    except ValueError:  
        print("Conversion failed!")  
        x = -1  
    except TypeError:  
        print("Conversion failed!")  
        x = -1  
    return x
```

```
'''A module for demonstrating exceptions.'''
```

```
def convert(s):  
    '''Convert to an integer.'''  
    x = -1  
    try:  
        x = int(s)  
        print("Conversion succeeded! x =", x)  
    except ValueError:  
        print("Conversion failed!")  
    except TypeError:  
        print("Conversion failed!")  
    return x
```

```
'''A module for demonstrating exceptions.'''
```

```
def convert(s):
```

```
    '''Convert to an integer.'''
```

```
    x = -1
```

```
    try:
```

```
        x = int(s)
```

```
        print("Conversion succeeded! x =", x)
```

```
    except (ValueError, TypeError):
```

```
        print("Conversion failed!")
```

```
    return x
```

```
'''A module for demonstrating exceptions.'''
```

```
def convert(s):  
    '''Convert to an integer.'''  
    x = -1  
    try:  
        x = int(s)  
    except (ValueError, TypeError):  
    return x
```

Exceptions for programmer errors

`IndentationError`

`SyntaxError`

`NameError`

Exceptions for programmer errors

IndentationError

SyntaxError

NameError

**You should not normally
catch these.**


```
'''A module for demonstrating exceptions.'''
```

```
def convert(s):  
    '''Convert to an integer.'''  
    x = -1  
    try:  
        x = int(s)  
    except (ValueError, TypeError):  
        pass  
    return x
```

```
'''A module for demonstrating exceptions.'''
```

```
def convert(s):  
    '''Convert to an integer.'''  
    try:  
        return int(s)  
    except (ValueError, TypeError):  
        return -1
```

```
'''A module for demonstrating exceptions.'''
```

```
def convert(s):  
    '''Convert to an integer.'''  
    try:  
        return int(s)  
    except (ValueError, TypeError) as e:  
        return -1
```

```
'''A module for demonstrating exceptions.'''
```

```
import sys
```

```
def convert(s):
```

```
    '''Convert to an integer.'''
```

```
    try:
```

```
        return int(s)
```

```
    except (ValueError, TypeError) as e:
```

```
        print("Conversion error: {}".format(str(e)),
```

```
              file=sys.stderr)
```

```
    return -1
```

```
from math import log
```

```
def string_log(s):  
    v = convert(s)  
    return log(v)
```

Exceptions can not be **ignored**.

Exceptions can not be **ignored**.

But error codes can...

```
def convert(s):  
    '''Convert to an integer.'''  
    try:  
        return int(s)  
    except (ValueError, TypeError) as e:  
        print("Conversion error: {}".format(str(e)),  
              file=sys.stderr)  
        raise
```


Exceptions are part of the API

Callers need to know **what**
exceptions to expect, and **when**.

Exceptions are part of the API

Callers need to know **what exceptions** to expect, and **when**.



```
def sqrt(x):  
    '''Compute square roots using the method of Heron of Alexandria.  
  
    Args:  
        x: The number for which the square root is to be computed.  
  
    Returns:  
        The square root of x.  
    '''  
    guess = x  
    i = 0  
    while guess * guess != x and i < 20:  
        guess = (guess + x / guess) / 2.0  
        i += 1  
    return guess  
  
def main():  
    print(sqrt(9))  
    print(sqrt(2))  
  
if __name__ == '__main__':  
    main()
```

```
def sqrt(x):  
    '''Compute square roots using the method of Heron of Alexandria.  
  
    Args:  
        x: The number for which the square root is to be computed.  
  
    Returns:  
        The square root of x.  
    '''  
    guess = x  
    i = 0  
    while guess * guess != x and i < 20:  
        guess = (guess + x / guess) / 2.0  
        i += 1  
    return guess  
  
def main():  
    print(sqrt(9))  
    print(sqrt(2))  
    print(sqrt(-1))  
  
if __name__ == '__main__':  
    main()
```

```
def sqrt(x):  
    '''Compute square roots using the method of Heron of Alexandria.  
  
    Args:  
        x: The number for which the square root is to be computed.  
  
    Returns:  
        The square root of x.  
    '''  
    guess = x  
    i = 0  
    while guess * guess != x and i < 20:  
        guess = (guess + x / guess) / 2.0  
        i += 1  
    return guess  
  
def main():  
    print(sqrt(9))  
    print(sqrt(2))  
    try:  
        print(sqrt(-1))  
    except ZeroDivisionError:  
        print("Cannot compute square root of a negative number.")  
    print("Program execution continues normally here.")  
  
if __name__ == '__main__':  
    main()
```

```
def sqrt(x):  
    '''Compute square roots using the method of Heron of Alexandria.  
  
    Args:  
        x: The number for which the square root is to be computed.  
  
    Returns:  
        The square root of x.  
    '''  
    guess = x  
    i = 0  
    while guess * guess != x and i < 20:  
        guess = (guess + x / guess) / 2.0  
        i += 1  
    return guess  
  
def main():  
    try:  
        print(sqrt(9))  
        print(sqrt(2))  
        print(sqrt(-1))  
        print("This is never printed.")  
    except ZeroDivisionError:  
        print("Cannot compute square root of a negative number.")  
  
    print("Program execution continues normally here.")  
  
if __name__ == '__main__':  
    main()
```

Use **exceptions** that **users will**
anticipate.

Standard exceptions are often
the best choice.

```
def sqrt(x):  
    '''Compute square roots using the method of Heron of Alexandria.
```

Args:

x: The number for which the square root is to be computed.

Returns:

The square root of x.

```
'''
```

```
guess = x
```

```
i = 0
```

```
try:
```

```
    while guess * guess != x and i < 20:
```

```
        guess = (guess + x / guess) / 2.0
```

```
        i += 1
```

```
except ZeroDivisionError:
```

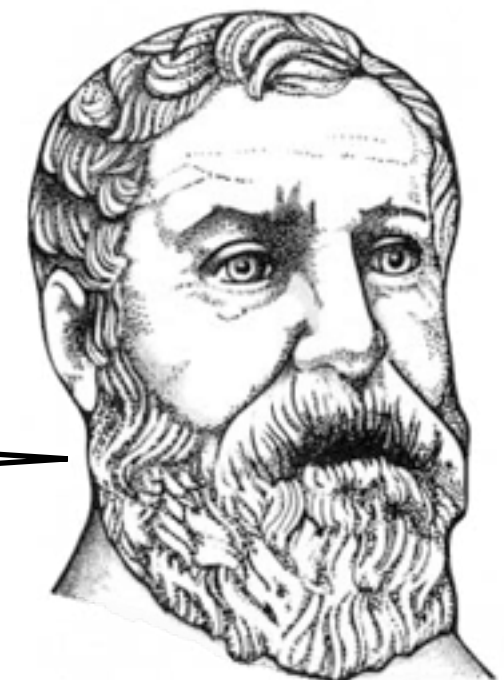
```
    raise ValueError()
```

```
return guess
```



```
def sqrt(x):  
    '''Compute square roots using the method of Heron of Alexandria.  
  
    Args:  
        x: The number for which the square root is to be computed.  
  
    Returns:  
        The square root of x.  
    '''  
    guess = x  
    i = 0  
    try:  
        while guess * guess != x and i < 20:  
            guess = (guess + x / guess) / 2.0  
            i += 1  
    except ZeroDivisionError:  
        raise ValueError()  
    return guess
```

Wasteful!



```
def sqrt(x):  
    '''Compute square roots using the method of Heron of Alexandria.  
  
    Args:  
        x: The number for which the square root is to be computed.  
  
    Returns:  
        The square root of x.  
  
    Raises:  
        ValueError: If x is negative.  
    '''
```

```
    if x < 0:  
        raise ValueError("Cannot compute square root "  
                           "of negative number {}".format(x))
```

```
    guess = x  
    i = 0  
    while guess * guess != x and i < 20:  
        guess = (guess + x / guess) / 2.0  
        i += 1  
    return guess
```

```
import sys
```

```
def main():
```

```
    try:
```

```
        print(sqrt(9))
```

```
        print(sqrt(2))
```

```
        print(sqrt(-1))
```

```
        print("This is never printed.")
```

```
    except ValueError as e:
```

```
        print(e, file=sys.stderr)
```

```
    print("Program execution continues normally here.")
```

Exceptions are part of the API

Exceptions are parts of **families of related functions** referred to at “**protocols**”.

Use **common or existing**
exception types **when possible.**

Use **common or existing**
exception types **when possible.**

IndexError

KeyError

ValueError

TypeError

etc...

Use **common or existing**
exception types **when possible.**

IndexError

KeyError

ValueError

TypeError

Follow existing usage
patterns.



IndexError

integer index is out of range



ValueError

object is of the right type, but contains an inappropriate value.



KeyError

Look-up in a mapping fails

Avoid protecting against
TypeErrors.

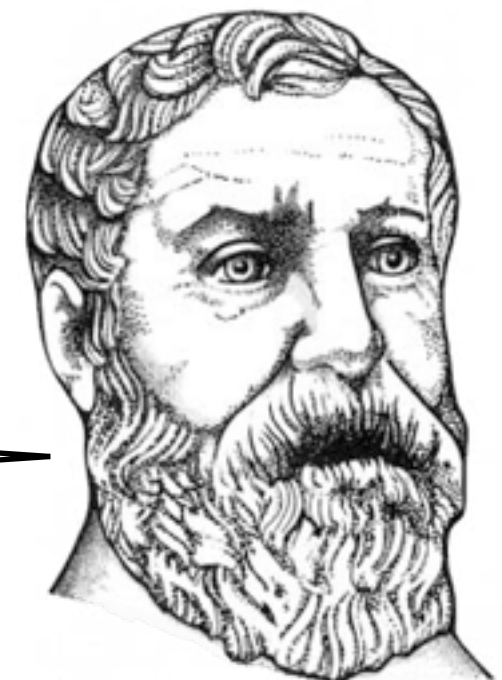
Avoid protecting against
TypeErrors.

This is generally “against
the grain” in Python.

```
def convert(s):  
    '''Convert to an integer.'''  
    if not isinstance(s, int):  
        raise TypeError(  
            "Argument must be a number".)  
  
    try:  
        return int(s)  
    except (ValueError, TypeError) as e:  
        print("Conversion error: {}".format(str(e)),  
            file=sys.stderr)  
        raise
```

```
def convert(s):  
    '''Convert to an integer.'''  
    if not isinstance(s, int):  
        raise TypeError(  
            "Argument must be a number".)  
  
    try:  
        return int(s)  
    except (ValueError, TypeError) as e:  
        print("Conversion error: {}".format(str(e)),  
            file=sys.stderr)  
        raise
```

Just let it fail!



It's usually **not worth checking**
types.

This can **limit** your functions
unnecessarily.

Dealing with failures

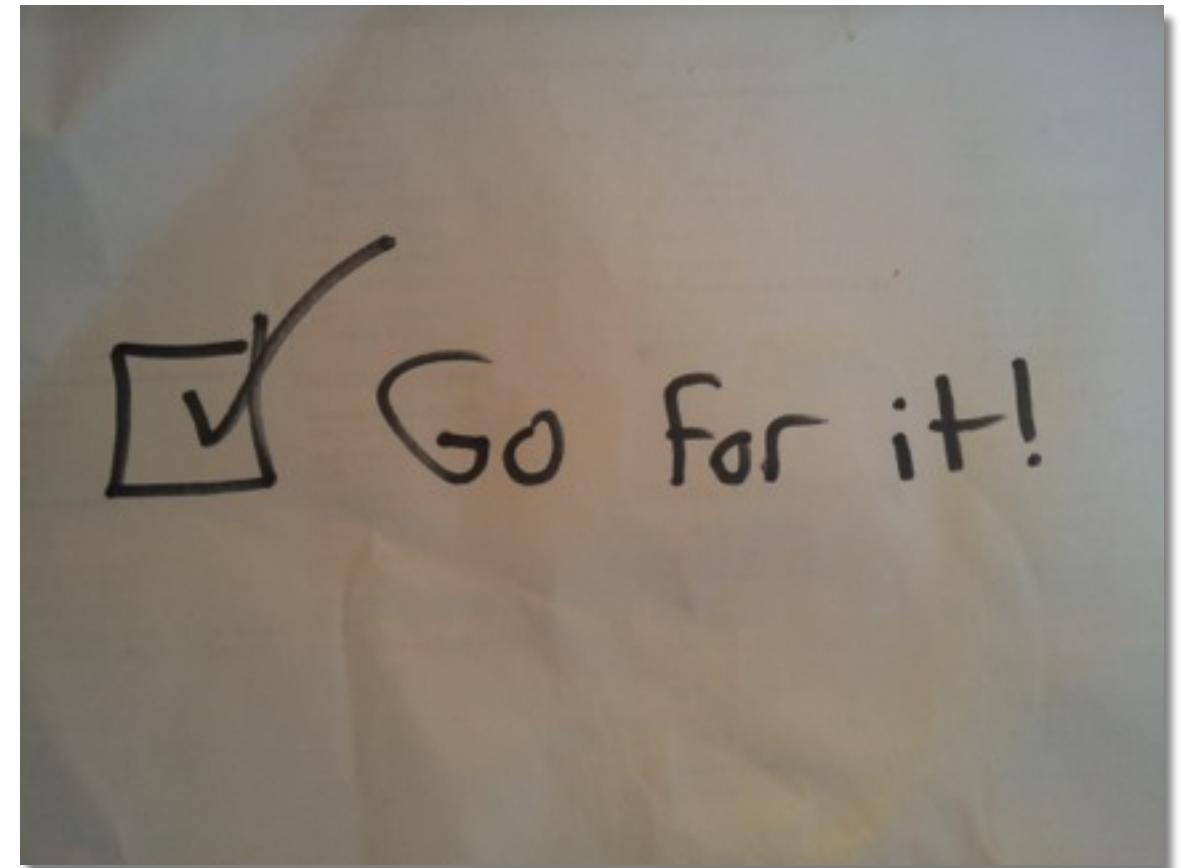
Cessna 60146

Preflight

A R O W

Remove Control Lock	✓ Leading Edge
✓ Ignition Off	✓ Cables & Bolts
Master ON	✓ Elevator & Rudder
Lower Flaps	Remove Tiedown
✓ Fuel Guages	✓ Leading Edge
Fuel On —	✓ Flaps
Master Off	✓ Weights & Hinges
✓ Tire and Brake	Remove Tiedown
✓ Tank for Water	✓ Leading Edge
✓ Fuel & Cap	✓ Tire & Brake
✓ Pitot Opening	✓ T & B for Water
✓ Overflow Opening	✓ Fuel & Cap
✓ Stall Opening	✓ Oil & Drain Str
Remove Tie Down	✓ Strut & Tire
✓ Leading Edge	✓ Prop Nicks/Sec
✓ Weights & Hinges	✓ Carb Filter
✓ Flaps	✓ Static Port

VS.



Two Philosophies

Look **B**efore **Y**ou **L**eap

vs.

It's **E**asier to **A**sk **F**orgiveness
than **P**ermission

Two Philosophies

It's **E**asier to **A**sk **F**orgiveness
than **P**ermission



```
import os
```

```
p = '/path/to/datafile.dat'
```

```
if os.path.exists(p):  
    process_file(p)
```

```
else:
```

```
    print('No such file as {}'.format(p))
```

```
import os
```

```
p = '/path/to/datafile.dat'
```

```
if os.path.exists(p):
```

```
    process_file(p)
```

```
else:
```

```
    print('No such file as {}'.format(p))
```

```
import os
```

```
p = '/path/to/datafile.dat'
```

```
if os.path.exists(p): Race condition  
    process_file(p)
```

```
else:
```

```
    print('No such file as {}'.format(p))
```

```
p = '/path/to/datafile.dat'

try:
    process_file(f)
except OSError as e:
    print('Could not process file because{}\n'
          .format(str(e)))
```

Local vs. Non-Local Handling

Error codes require interspersed, **local handling**.

Exceptions allow centralized, **non-local handling**.

EAFP + Exceptions

Exceptions require explicit
handling.

Error codes are **silent** by default.

EAFP + Exceptions

Exceptions require explicit
handling.

Error codes are **silent** by default.

EAFP + Exceptions = errors
are difficult to ignore!

Resource Cleanup with Finally

`try...finally` lets you clean up
whether an **exception occurs or not.**

```
import os
```

```
def make_at(path, dir_name):  
    original_path = os.getcwd()  
    os.chdir(path)  
    os.mkdir(dir_name)  
    os.chdir(original_path)
```

```
import os
```

```
def make_at(path, dir_name):  
    original_path = os.getcwd()  
    os.chdir(path)  
    os.mkdir(dir_name)  
    os.chdir(original_path)
```

If this fails...



```
import os
```

```
def make_at(path, dir_name):  
    original_path = os.getcwd()  
    os.chdir(path)  
    os.mkdir(dir_name)  
    os.chdir(original_path)
```

If this fails...



...then this won't happen!



```
import os

def make_at(path, dir_name):
    original_path = os.getcwd()
    try:
        os.chdir(path)
        os.mkdir(dir_name)
    finally:
        os.chdir(original_path)
```

```
import os

def make_at(path, dir_name):
    original_path = os.getcwd()
    try:
        os.chdir(path)
        os.mkdir(dir_name)
    finally:
        os.chdir(original_path)
```

```
import os

def make_at(path, dir_name):
    original_path = os.getcwd()
    try:
        os.chdir(path)
        os.mkdir(dir_name)
    finally:
        os.chdir(original_path)
```

**finally-block is
executed no matter how
the try-block exits.**


```
import os
import sys

def make_at(path, dir_name):
    original_path = os.getcwd()
    try:
        os.chdir(path)
        os.mkdir(dir_name)
    except OSError as e:
        print(e, file=sys.stderr)
        raise
    finally:
        os.chdir(original_path)
```

```
import os
import sys

def make_at(path, dir_name):
    original_path = os.getcwd()
    try:
        os.chdir(path)
        os.mkdir(dir_name)
    except OSError as e:
        print(e, file=sys.stderr)
        raise
    finally:
        os.chdir(original_path)
```



Runs even if `OSError` is
thrown and handled.

Moment of Zen

Errors should never
pass silently, unless
explicitly silenced.

Errors are like bells
And if we make them silent
They are of no use



Platform-Specific Modules



Windows

msvcrt



Linux

OSX

sys

tty

termios

```
"""keypress - A module for detecting a single keypress."""
```

```
try:
```

```
    import msvcrt
```

```
    def getkey():
```

```
        """Wait for a keypress and return a single character string."""
```

```
        return msvcrt.getch()
```

```
except ImportError:
```

```
    import sys
```

```
    import tty
```

```
    import termios
```

```
    def getkey():
```

```
        """Wait for a keypress and return a single character string."""
```

```
        fd = sys.stdin.fileno()
```

```
        original_attributes = termios.tcgetattr(fd)
```

```
        try:
```

```
            tty.setraw(sys.stdin.fileno())
```

```
            ch = sys.stdin.read(1)
```

```
        finally:
```

```
            termios.tcsetattr(fd, termios.TCSADRAIN, original_attributes)
```

```
        return ch
```

```
# If either of the Unix-specific tty or termios are not found,
```

```
# we allow the ImportError to propagate from here
```

"""keypress - A module for detecting a single keypress."""

```
try:
    import msvcrt

    def getkey():
        """Wait for a keypress and return a single character string."""
        return msvcrt.getch()

except ImportError:

    import sys
    import tty
    import termios

    def getkey():
        """Wait for a keypress and return a single character string."""
        fd = sys.stdin.fileno()
        original_attributes = termios.tcgetattr(fd)
        try:
            tty.setraw(sys.stdin.fileno())
            ch = sys.stdin.read(1)
        finally:
            termios.tcsetattr(fd, termios.TCSADRAIN, original_attributes)
        return ch

    # If either of the Unix-specific tty or termios are not found,
    # we allow the ImportError to propagate from here
```

```
"""keypress - A module for detecting a single keypress."""
```

```
try:
```

```
    import msvcrt
```

```
    def getkey():
```

```
        """Wait for a keypress and return a single character string."""
```

```
        return msvcrt.getch()
```

```
except ImportError:
```

```
    import sys
```

```
    import tty
```

```
    import termios
```

```
    def getkey():
```

```
        """Wait for a keypress and return a single character string."""
```

```
        fd = sys.stdin.fileno()
```

```
        original_attributes = termios.tcgetattr(fd)
```

```
        try:
```

```
            tty.setraw(sys.stdin.fileno())
```

```
            ch = sys.stdin.read(1)
```

```
        finally:
```

```
            termios.tcsetattr(fd, termios.TCSADRAIN, original_attributes)
```

```
        return ch
```

```
# If either of the Unix-specific tty or termios are not found,
```

```
# we allow the ImportError to propagate from here
```

```
"""keypress - A module for detecting a single keypress."""
```

```
try:
```

```
    import msvcrt
```

```
    def getkey():
```

```
        """Wait for a keypress and return a single character string."""
```

```
        return msvcrt.getch()
```

```
except ImportError:
```

```
import sys
```

```
import tty
```

```
import termios
```

```
def getkey():
```

```
    """Wait for a keypress and return a single character string."""
```

```
    fd = sys.stdin.fileno()
```

```
    original_attributes = termios.tcgetattr(fd)
```

```
    try:
```

```
        tty.setraw(sys.stdin.fileno())
```

```
        ch = sys.stdin.read(1)
```

```
    finally:
```

```
        termios.tcsetattr(fd, termios.TCSADRAIN, original_attributes)
```

```
    return ch
```

```
# If either of the Unix-specific tty or termios are not found,
```

```
# we allow the ImportError to propagate from here
```



```
"""keypress - A module for detecting a single keypress."""
```

```
try:
```

```
    import msvcrt
```

```
    def getkey():
```

```
        """Wait for a keypress and return a single character string."""
```

```
        return msvcrt.getch()
```

```
except ImportError:
```

```
    import sys
```

```
    import tty
```

```
    import termios
```

```
    def getkey():
```

```
        """Wait for a keypress and return a single character string."""
```

```
        fd = sys.stdin.fileno()
```

```
        original_attributes = termios.tcgetattr(fd)
```

```
        try:
```

```
            tty.setraw(sys.stdin.fileno())
```

```
            ch = sys.stdin.read(1)
```

```
        finally:
```

```
            termios.tcsetattr(fd, termios.TCSADRAIN, original_attributes)
```

```
        return ch
```

```
# If either of the Unix-specific tty or termios are not found,  
# we allow the ImportError to propagate from here
```

```
"""keypress - A module for detecting a single keypress."""
```

```
try:
```

```
    import msvcrt
```

```
    def getkey():
```

```
        """Wait for a keypress and return a single character string."""
```

```
        return msvcrt.getch()
```

```
except ImportError:
```

```
    import sys
```

```
    import tty
```

```
    import termios
```

```
    def getkey():
```

```
        """Wait for a keypress and return a single character string."""
```

```
        fd = sys.stdin.fileno()
```

```
        original_attributes = termios.tcgetattr(fd)
```

```
        try:
```

```
            tty.setraw(sys.stdin.fileno())
```

```
            ch = sys.stdin.read(1)
```

```
        finally:
```

```
            termios.tcsetattr(fd, termios.TCSADRAIN, original_attributes)
```

```
        return ch
```

```
# If either of the Unix-specific tty or termios are not found,
```

```
# we allow the ImportError to propagate from here
```

```
"""keypress - A module for detecting a single keypress."""
```

```
try:
```

```
    import msvcrt
```

```
    def getkey():
```

```
        """Wait for a keypress and return a single character string."""
```

```
        return msvcrt.getch()
```

```
except ImportError:
```

```
    import sys
```

```
    import tty
```

```
    import termios
```

```
    def getkey():
```

```
        """Wait for a keypress and return a single character string."""
```

```
        fd = sys.stdin.fileno()
```

```
        original_attributes = termios.tcgetattr(fd)
```

```
        try:
```

```
            tty.setraw(sys.stdin.fileno())
```

```
            ch = sys.stdin.read(1)
```

```
        finally:
```

```
            termios.tcsetattr(fd, termios.TCSADRAIN, original_attributes)
```

```
        return ch
```

```
# If either of the Unix-specific tty or termios are not found,
```

```
# we allow the ImportError to propagate from here
```

```
"""keypress - A module for detecting a single keypress."""
```

```
try:
```

```
    import msvcrt
```

```
    def getkey():
```

```
        """Wait for a keypress and return a single character string."""
```

```
        return msvcrt.getch()
```

```
except ImportError:
```

```
    import sys
```

```
    import tty
```

```
    import termios
```

```
    def getkey():
```

```
        """Wait for a keypress and return a single character string."""
```

```
        fd = sys.stdin.fileno()
```

```
        original_attributes = termios.tcgetattr(fd)
```

```
        try:
```

```
            tty.setraw(sys.stdin.fileno())
```

```
            ch = sys.stdin.read(1)
```

```
        finally:
```

```
            termios.tcsetattr(fd, termios.TCSADRAIN, original_attributes)
```

```
        return ch
```

```
# If either of the Unix-specific tty or termios are not found,
```

```
# we allow the ImportError to propagate from here
```



python

Exception Handling – Summary



python

Exception Handling – Summary

- Raising an exception interrupts normal program flow and transfers control to an exception handler.



python

Exception Handling – Summary

- Raising an exception interrupts normal program flow and transfers control to an exception handler.
- Exception handlers defined using the `try...except` construct.



python

Exception Handling – Summary

- Raising an exception interrupts normal program flow and transfers control to an exception handler.
- Exception handlers defined using the `try...except` construct.
- `try` blocks define a context for detecting exceptions.



python

Exception Handling – Summary

- Raising an exception interrupts normal program flow and transfers control to an exception handler.
- Exception handlers defined using the `try...except` construct.
- `try` blocks define a context for detecting exceptions.
- Corresponding `except` blocks handle specific exception types.



python

Exception Handling – Summary

- Raising an exception interrupts normal program flow and transfers control to an exception handler.
- Exception handlers defined using the `try...except` construct.
- `try` blocks define a context for detecting exceptions.
- Corresponding `except` blocks handle specific exception types.
- Python uses exceptions pervasively.
 - Many built-in language features depend on them.



python

Exception Handling – Summary

- Raising an exception interrupts normal program flow and transfers control to an exception handler.
- Exception handlers defined using the `try...except` construct.
- `try` blocks define a context for detecting exceptions.
- Corresponding `except` blocks handle specific exception types.
- Python uses exceptions pervasively.
 - Many built-in language features depend on them.
- `except` blocks can capture an exception, which are often of a standard type.



python

Exception Handling – Summary

- Raising an exception interrupts normal program flow and transfers control to an exception handler.
- Exception handlers defined using the `try...except` construct.
- `try` blocks define a context for detecting exceptions.
- Corresponding `except` blocks handle specific exception types.
- Python uses exceptions pervasively.
 - Many built-in language features depend on them.
- `except` blocks can capture an exception, which are often of a standard type.
- Programmer errors should not normally be handled.



python

Exception Handling – Summary

- Raising an exception interrupts normal program flow and transfers control to an exception handler.
- Exception handlers defined using the `try...except` construct.
- `try` blocks define a context for detecting exceptions.
- Corresponding `except` blocks handle specific exception types.
- Python uses exceptions pervasively.
 - Many built-in language features depend on them.
- `except` blocks can capture an exception, which are often of a standard type.
- Programmer errors should not normally be handled.
- Exceptional conditions can be signaled using `raise`.



python

Exception Handling – Summary

- Raising an exception interrupts normal program flow and transfers control to an exception handler.
- Exception handlers defined using the `try...except` construct.
- `try` blocks define a context for detecting exceptions.
- Corresponding `except` blocks handle specific exception types.
- Python uses exceptions pervasively.
 - Many built-in language features depend on them.
- `except` blocks can capture an exception, which are often of a standard type.
- Programmer errors should not normally be handled.
- Exceptional conditions can be signaled using `raise`.
- `raise` without an argument re-raises the current exception.



python

Exception Handling – Summary

- Generally do not check for `TypeError`s.



python

Exception Handling – Summary

- Generally do not check for `TypeError`s.
- Exception objects can be converted to strings using `str()`.



python

Exception Handling – Summary

- Generally do not check for `TypeError`s.
- Exception objects can be converted to strings using `str()`.
- A function's exceptions form part of its API.
 - They should be documented properly.



python

Exception Handling – Summary

- Generally do not check for `TypeError`s.
- Exception objects can be converted to strings using `str()`.
- A function's exceptions form part of its API.
 - They should be documented properly.
- Prefer to use built-in exception types when possible.



python

Exception Handling – Summary

- Generally do not check for `TypeError`s.
- Exception objects can be converted to strings using `str()`.
- A function's exceptions form part of its API.
 - They should be documented properly.
- Prefer to use built-in exception types when possible.
- Use the `try...finally` construct to perform cleanup actions.
 - May be used in conjunction with `except` blocks.

 python

Exception Handling – Summary

- **Output of `print()` can be redirected using the optional file argument.**

 python

Exception Handling – Summary

- **Output of `print()` can be redirected using the optional `file` argument.**
- **Use `and` or `or` for combining boolean expressions.**

 python

Exception Handling – Summary

- **Output of `print()` can be redirected using the optional `file` argument.**
- **Use `and` or `or` for combining boolean expressions.**
- **Return codes are too easily ignored.**

python **Exception Handling – Summary**

- **Output of `print()` can be redirected using the optional `file` argument.**
- **Use `and` or `or` for combining boolean expressions.**
- **Return codes are too easily ignored.**
- **Platform-specific actions can be implemented using `EAFP` along with catching `ImportErrors`.**